

CANVAS

DRAW ON THE WEB

AIRING

目錄

CANVAS

如果您喜欢本书，请使用支付宝扫描下面的二维码捐助作者。



谢谢您的支持！也欢迎您订阅本书。

如果书中有疏漏或错误之处，敬请您指出，期待您的pull request。如果有任何疑问也可以发送issue。

本书GitBook：https://airingursb.gitbooks.io/canvas/

本人邮箱：airing@ursb.me

本人博客：http://ursb.me

本书主页：http://airing.coding.me/canvas

本书GitHub：http://github.com/airingursb/canvas



CANVAS

DRAW ON THE WEB

AIRING

Ch1 HTML5简介

前言

今后的一个月内会连载详细的Canvas教程，从零基础开始，到Canvas API，再到基本动画与高级动画的实现，还会介绍视音频的处理、移动应用，最后如果有时间会扩展说一说3D、多人应用、游戏制作等。所以本课程虽说是Canvas教程，但其实就是详细的介绍Canvas API，之后基于Canvas实现其他更高级的功能。

如果你学过HTML4，或者CSS、Javascript，那么相信你入手起来会很快；如果你啥都没学过，属于零基础，那就更好了。因为你保有对这个未知领域的好奇心，这一切都会激发你更加努力的向前。而且零基础的童鞋也不用担心，本教程会在用到其他知识的时候会有详细的扩展说明，以Canvas为线索，学完它你基本上一系列的知识也都学会了，买一送一简直不能更赚了！其中也会穿插讲解数学、物理学、运动学的一些简单的知识，每个知识点都会提供案例，每个案例都会提供页面演示，源文件可以去我托管在**github**上的一个开源项目上下载。

> 提示：本教程中有链接的地方都不妨点一点 ：）

好了，是不是摩拳擦掌、迫不及待准备上了？那就让我们开始走进HTML5的世界吧！

HTML5介绍

HTML5是新一代的HTML(Hyper Text Markup Language)，即超文本标记语言，于去年10月28日正式发布，它是全新的、互联网上构建页面的标准语言。

那么究竟什么是HTML5？在W3C HTML5的常见问题中，关于HTML5是这样说明的：*HTML5*是一个开放的平台下开发的免费许可条款。

具体来说，对这句话有以下两种理解：

- 指一组共同构成了未来开放式网络平台的技术。这些技术包括HTML5规范、CSS3、SVG、MATHML、地理位置、XmlHttpRequest、Context 2D、Web字体以及其他技术。这一套技术的边界是非正式的，且随时间变化的。
- 指**HTML5**规范，当然也是开放式网络平台的一部分。

Canvas的浏览器支持



由于课程的主要内容是Canvas，以下我列出了最流行的Web浏览器以及它们开始支持Canvas元素的最小版本号。

Safari	Firefox	IE	Chrome	Opear	iOS Safari	Android Brower
3.2	3.5	9	9	10.6	3.2	2.1

这里我推荐使用Chrome。

所以在学习本课程之前，赶快给你的电脑装上最新版的Chrome吧！

基础的HTML5页面

简单的HTML5页面

```
<!doctype html>
<html lang="zh">
<head>
<meta charset="UTF-8">
<title>基础的HTML5页面</title>
</head>
<body>
Hello Airing!
</body>
</html>
```

演示 1-1

运行结果如下：





HTML是由一个个形如尖括号 `<>` 的标签元素组成，这些标签通常是成对出现，并且标签之间只能嵌套不能交叉。

> 扩展：
>
> 成对出现的叫做闭合标签，单个出现的叫做单标签。不管怎样都是闭合的(单标签可以不闭合，但是在XHTML中严格要求了闭合)。闭合标签又分为开始标签和结束标签，如 `<body>` 是开始标签， `</body>` 是结束标签。自标签如 `<input/>` `<br/>` 等。
>
> 关于更多的标签，建议大家自行了解一下。推荐W3school平台自学。

这里我们着重讲一下上述代码中出现的标签。

`<!doctype html>`

这个标签说明 Web 浏览器将在标准模式下呈现页面。根据 W3C 定义的 HTML5 规范，这是 HTML5 文档所必需的。这个标签简化了长期以来在不同的浏览器呈现 HTML 页面时出现的奇怪差异。它通常为文档中的第一行。

`<html lang="en">`

这是包含语言说明的标签，例如，"en"为英语，"zh"为中文。

`<head>` ... `</head>`

这2个标记符分别表示头部信息的开始和结尾。头部中包含的标记是页面的标题、序言、说明等内容，它本身不作为内容来显示，但影响网页显示的效果。头部中最常用的标记符是 `<title>` 标记符和 `<meta>` 标记符。

> 以下表格列出了HTML head 元素下的所有标签和功能：
>
> | 标签 | 描述 |
> | --- | --- |
> | `<head>` | 定义了文档的信息 |
> | `<title>` | 定义了文档的标题 |
> | `<base>` | 定义了页面链接标签的默认链接地址 |
> | `<link>` | 定义了一个文档和外部资源之间的关系 |
> | `<meta>` | 定义了HTML文档中的元数据 |
> | `<script>` | 定义了客户端的脚本文件 |
> | `<style>` | 定义了HTML文档的样式文件 |

`<meta charset="UTF-8">`

这个标签说明 Web 浏览器使用的字符编码模式，这里通常设置为UTF-8。如果没有需要特别设置的没必要改变它。这也是 HTML5 页面需要的元素。

`<title>` ... `</title>`

这个标签说明在浏览器窗口展示的 HTML 的标题。这是一个很重要的标记，它是搜索引擎用来在 HTML 页面上收录内容的主要信息之一。

`<body>` ... `</body>`

网页中显示的实际内容均包含在这2个 `<body>` 之间。

综上，HTML5网页是由第一行的 `<!doctype html>` 与 `<html>` 部分组成，而 `<html>` 主要分为两部分——由 `<head>` 标签规定的头部部分，和由 `<body>` 规定的主体部分。

这样，我们就把最简单的HTML网页的基本结构给捋出来了。

好的，接下来就让我们的主角Canvas登场吧！不过，在此之前，建议大家自行了解一下HTML的常用标签及其功能~

Ch2 开始前的准备

添加一个 Canvas

在HTML中添加Canvas非常简单，只需要在HTML的 `<body>` 部分，添加上 `<canvas>` 标签就可以了！可以参考下面的代码。

```
<!doctype html>
<html lang="zh">
<head>
<meta charset="UTF-8">
<title>基础的HTML5页面</title>
</head>

<body>
    <canvas id="canvas">
    你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</body>

</html>
```

演示 2-1

由于结果页面是一个完完全全的空白页面，所以这里我就不贴图了。大家可能会很好奇，为什么会是一个空白呢？（废话，我还没来得及画画呢！）Canvas的本意是画布，也就是画布的意思（废话...），画布在HTML5中是透明的，是不可见的。

那 `<canvas>` 标签中的那段文本是什么意思呢？那是一旦浏览器执行HTML页面时不支持Canvas，就会显示这段文字，换言之，只要你的浏览器支持Canvas，页面上就不会显示这个文本。

那 `<canvas>` 中的id是什么意思？id是标签的属性之一，在JavaScript代码中用来指定特定的 `<canvas>` 的名字，就像一个人的身份证号码一样，是唯一的。

为了更清楚的展示Canvas，以及方便之后的演示，我稍微修改了一下代码，之后的绘图都会在这个Canvas上绘制。

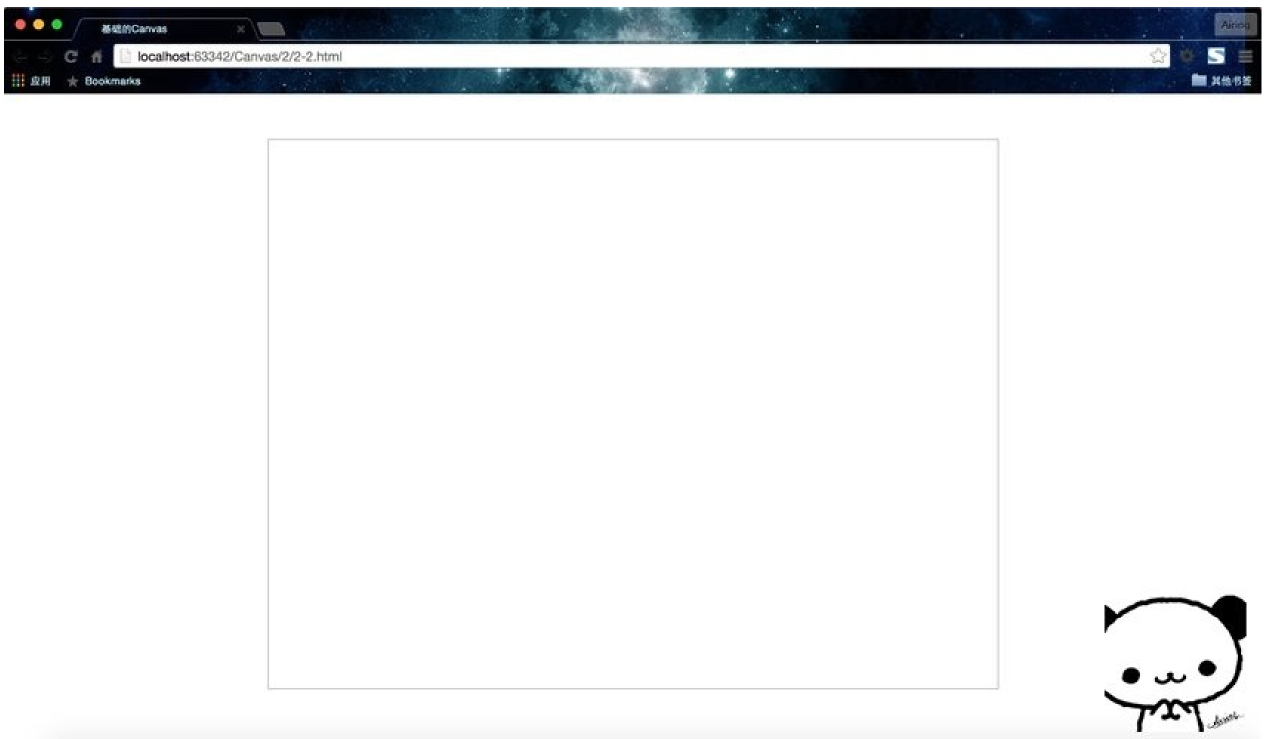
```
<!doctype html>
<html lang="zh">
<head>
<meta charset="UTF-8">
<title>基础的Canvas</title>
</head>

<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; display: block; margin: 50px auto;" width="800" height="600">
    你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>
</body>

</html>
```

演示 2-2

运行结果：



对以上代码有几点说明：

1. 添加了 `<div>` 标签，将 `<canvas>` 包裹其中，个人习惯，暂时并没有什么卵用。
2. 给 `<canvas>` 标签指定了width和height属性，规定了它的宽和高。
3. 给 `<canvas>` 标签添加了一个内联样式，使其变为块级元素并居中显示。

> 关于CSS的内容这里不做说明，毕竟这不是本课程的主角，若做扩展会花费大量篇幅。考虑到HTML、CSS是基础，建议大家花点时间自己学，推荐慕课网的HTML+CSS课程）

引用 Canvas 元素

文档对象模型(DOM)

文档对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展标志语言的标准编程接口。Document Object Model的历史可以追溯至1990年代后期微软与Netscape的“浏览器大战”，双方为了在JavaScript与JScript一决生死，于是大规模的赋予浏览器强大的功能。微软在网页技术上加入了不少专属事物，计有VBScript、ActiveX、以及微软自家的DHTML格式等，使不少网页使用非微软平台及浏览器无法正常显示。DOM即是当时蕴酿出来的杰作。

文档对象模型代表了在 **HTML** 页面上的所有对象。它是语言中立且平台中立的。它允许页面的内容和样式被 **Web** 浏览器渲染之后再次更新。用户可以通过 **JavaScript** 访问 **DOM**。

在开始使用 `<canvas>` 前,首先需要了解两个特定的 DOM 对象：window 和 document。

- window 对象是 DOM 的最高一级，需要对这个对象进行检测来确保开始使用 Canvas 应用程序之前，已经加载了所有的资源和代码。
- document 对象包含所有在 HTML 页面上的 HTML 标签。需要对这个对象进行检索来找 出用 JavaScript 操纵 `<canvas>` 的实例。

JavaScript 放置位置

使用 JavaScript 为 Canvas 编程会产生一个问题：在创建的页面中，从哪里启动JavaScript程序?

把 JavaScript 放进 HTML 页面的 `<head>` 标签中是个不错的主意，这样做的好处是很容易找到它，也是上一章我们介绍 `<head>` 中所提到的。但是，把 JavaScript 程序放在这里就意味着整个 HTML 页面要加载完 JavaScrpit 才能配合 HTML 运行，这段 JavaScript 代码也会在整个页面加载前就开始执行了。结果就是，运行 JavaScript 程序之前必须检查 HTML 页面是否已经加载完毕。

最近有一个趋势是将 JavaScript 放在 HTML 文档结尾处的 `</body>` 标签之前，这样就可以确保在 JavaScript 运行时整个页面已经加载完毕。然而，由于在运行 `<canvas>` 程序前需要使用 JavaScript 测试页面是否加载，因此最好还是将 JavaScript 放在 `<head>` 中。

不过本人不走寻常路(笑)，所以之后的案例，还是按照自己的编码风格将JavaScript代码放在了 `<body>` 的尾部。当然，如果JavaScript代码有些多，就推荐使用加载外部 .js 文件的方式。代码大致如下：

```
<script type="text/javascript" src="bootstarp.js"></script>
```

在实际项目开发中，都是将HTML、CSS、JS三者完全分离的。不过用于案例演示代码略少，所以大多没有使用加载外部 .js 文件的方式。

获取canvas对象

获取canvas对象其实就是一句话的事情。

```
var canvas = document.getElementById("canvas");
```

`var` 用于变量定义，由于JS是弱类型语言，所以定义啥变量都用 `var` 。跟在 `var` 之后的canvas是变量。使用 `document` 对象的 `getElementById()` 的方法，通过id获取对象。之前我们为 `<canvas>` 标签赋予了一个id，名叫canvas，所以该句话最后一个canvas是指 `<canvas>` 的id——canvas。（是不是有点绕，需要自己多读几遍捋清楚。）

获得画笔(2D环境)

画画首先需要啥？画笔啊。获取canvas画笔也是一句话的事情，就是直接使用刚才获得的canvas对象，调用它的 `getContext("2d")` 方法，即可。

```
var context = canvas.getContext("2d");
```

这里的context便是画笔了。

在其他教程中都是使用2D环境这个专有术语，我觉得画笔更加形象。灵感引自Java中Graphics类的g画笔，原理与之相同。

总结

准备工作只有三步：

1. 布置画布：通过添加 `<canvas>` 标签，添加**canvas**元素
2. 获取画布：通过 `<canvas>` 标签的**id**，获得**canvas**对象
3. 获得画笔：通过**canvas**对象的 `getContext("2d")` 方法，获得**2D**环境

对应的代码也就是三句话：

1. `<canvas id=“canvas”></canvas>`
2. `var canvas = document.getElementById("canvas");`
3. `var context = canvas.getContext("2d");`

完整代码如下。

```
<!doctype html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>基础的Canvas</title>
</head>

<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; display: block; margin: 50px auto;" width="800" height="600">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
window.onload = function(){
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
}
</script>
</body>

</html>
```

演示 2-3

注意几点：

1. JavaScript代码需要包裹在 `<script>` 标签中。
2. `window.onload = function(){}` 加载页面后就要立即执行，表示网页加载完执行后面的那个function函数体的内容。

至此，画布和画笔已经准备完毕，接下来就让我们使用画笔(context对象)绘制出高逼格的图像吧！觉醒吧！艺术家之魂！

Ch3 从线段开始

怎么画线段？

上一讲我们已经得到了咱们的画布和画笔，在发挥艺术家之魂前，还是要像小孩牙牙学语一样，我们也得从画一条线段开始。因为画线段是最简单的，最基础的。但是别小看了它。下面是我从度娘那里找到的一个由线条组成的图像。



是不是很有魔性？

言归正传。怎么画线条？和现实中画画差不多：

- 移动画笔，使画笔移动至绘画的开始处
- 确定第一笔的停止点
- 规划好之后，选择画笔（包括画笔的粗细和颜色等）
- 确定绘制

因为 **Canvas**是基于状态的绘制（很重要，后面会解释），所以前面几步都是在确定状态，最后一步才会具体绘制。

移动画笔(moveTo())

之前我们获得了画笔 `context` ，所以以此为例，给出改方法的使用实例——`context.moveTo(100,100)` 。这句代码的意思是 移动画笔至**(100,100)**这个点（单位是**px**）。记住，这里是以 `canvas` 画布的左上角为笛卡尔坐标系的原点，且y轴的正方向向下，x轴的正方向向右。

笔画停点(lineTo())

同理，`context.lineTo(600,600)` 。这句的意思是从 上一笔的停止点 绘制到(600,600)这里。不过要清楚，这里的 `moveTo()``lineTo()` 都只是状态而已，是规划，是我准备要画，还没有开始画，只是一个计划而已！

选择画笔

这里我们暂且只设置一下画笔的颜色和粗细。

`context.lineWidth = 5` ，这句话的意思是设置画笔(线条)的粗细为 5 px。

`context.strokeStyle = "#AA394C"` ，这句话的意思是设置画笔(线条)的颜色为玫红色。

因为Canvas是基于状态的绘制，所以我们在选择画笔粗细和颜色的同时，其实也是选择了线条的粗细和颜色。

确定绘制

确定绘制只有两种方法，`fill()` 和 `stroke()` ，有点绘画基础的应该知道，前者是指填充，后者是指描边。因为我们只是绘制线条，所以只要描边就可以了。调用代码 `context.stroke()` 即可。

画一个线条

不就一条线段吗！废话了这么多！那我们就开始画吧。

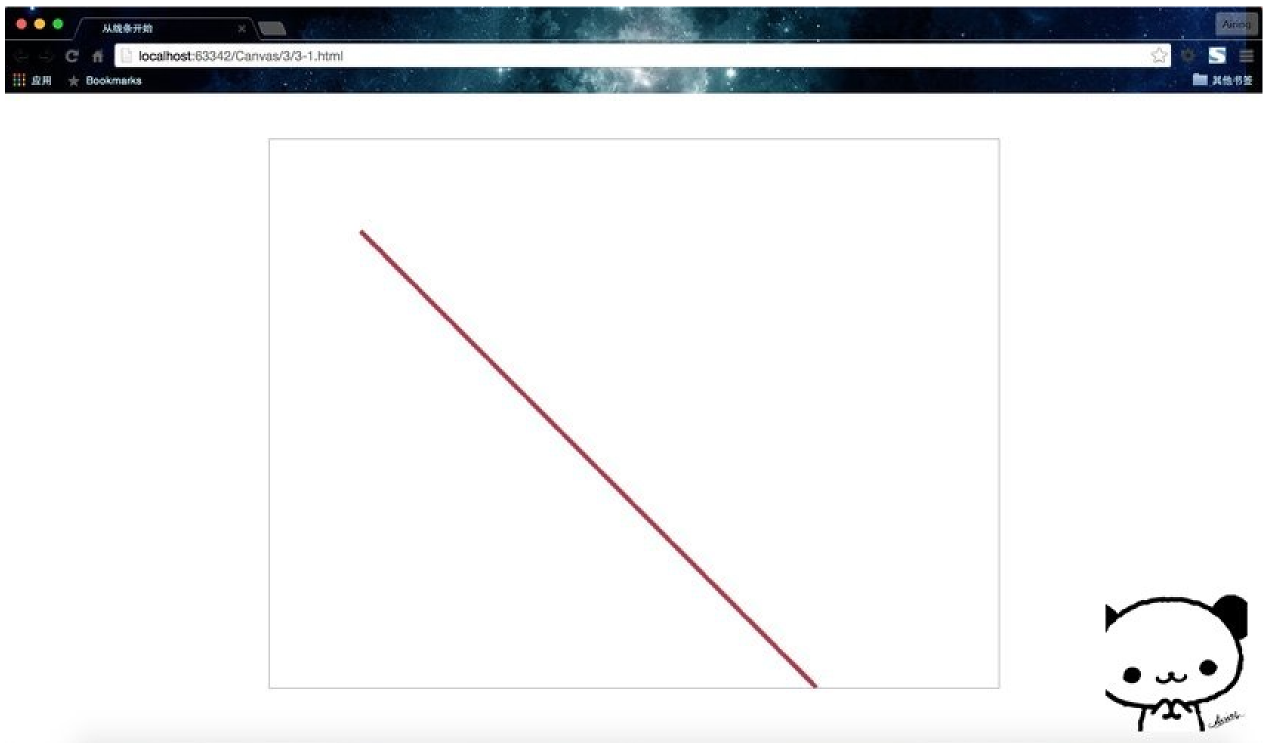
```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>从线条开始</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.moveTo(100,100);
        context.lineTo(600,600);
        context.lineWidth = 5;
        context.strokeStyle = "#AA394C";
        context.stroke();
    }
</script>
</body>
</html>
```

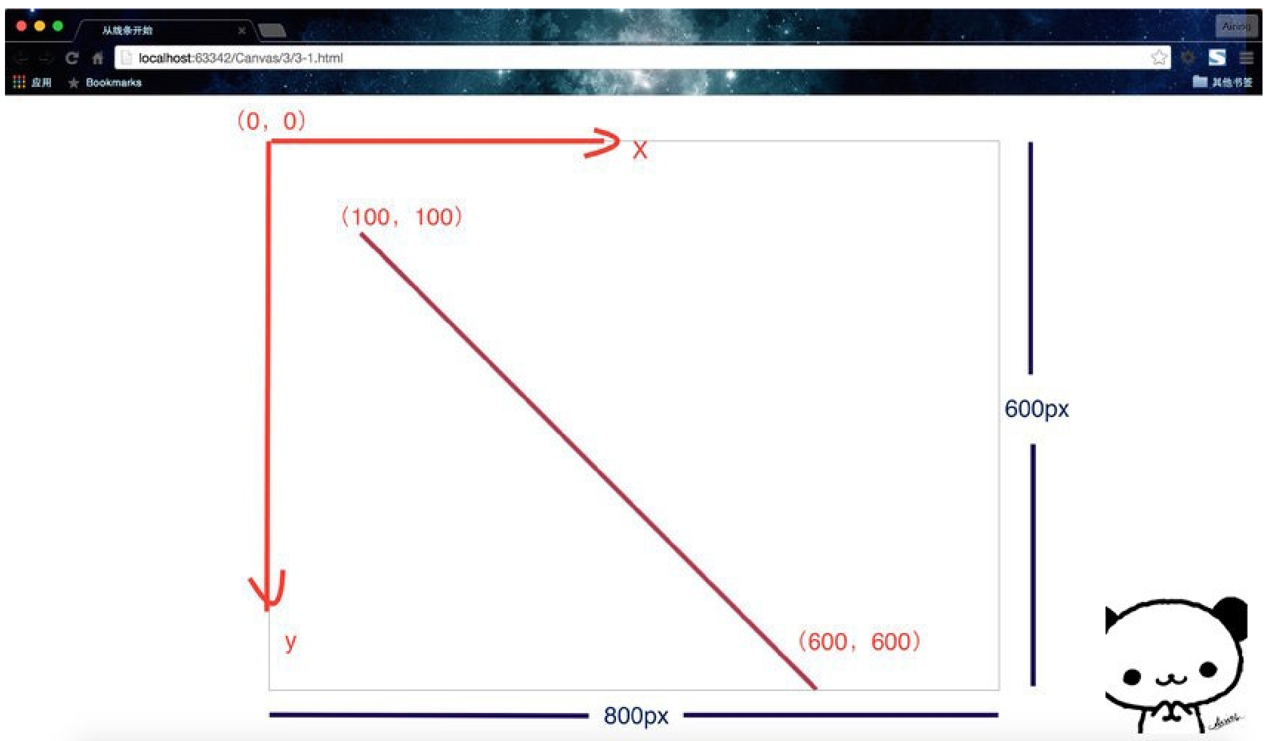
演示 3-1

运行结果：



(一直有小伙伴问我页面右下角的熊是什么鬼？哦哦，之前忘解释了，那个是我的防伪水印！😄)

我还标注了一个页面解析图，供大家参考。



这里我将原本 `<canvas>` 标签中的 `width` 和 `height` 去掉了，但在JavaScript代码中设置了 `canvas` 对象的 `width` 和 `height` 的属性。

> 小结：要设置画布的大小，只有这两种方法
>
> 1. 在 `<canvas>` 标签中设置；
> 2. 在JS代码中设置 `canvas` 的属性

怎么样，是不是非常的酷。接下来我们要加快脚步了，绘制一个多线条组成的图形。是不是感觉自己离艺术家又进了一步呢？别看这只是简简单单的一条线段，这一画只是我们的一小步，但却是人类的一大步！😇

Ch4 多线条组成图形

绘制折线

上一节中，我们已经成功绘制了一条线段。那么，如果我要绘制有两个笔画甚至是很多笔画的折线怎么办呢？

聪明的小伙伴肯定已经想到了，这还不简单，复用 `lineTo()` 就可以了。下面我就献丑随便画了一条优美的折线~

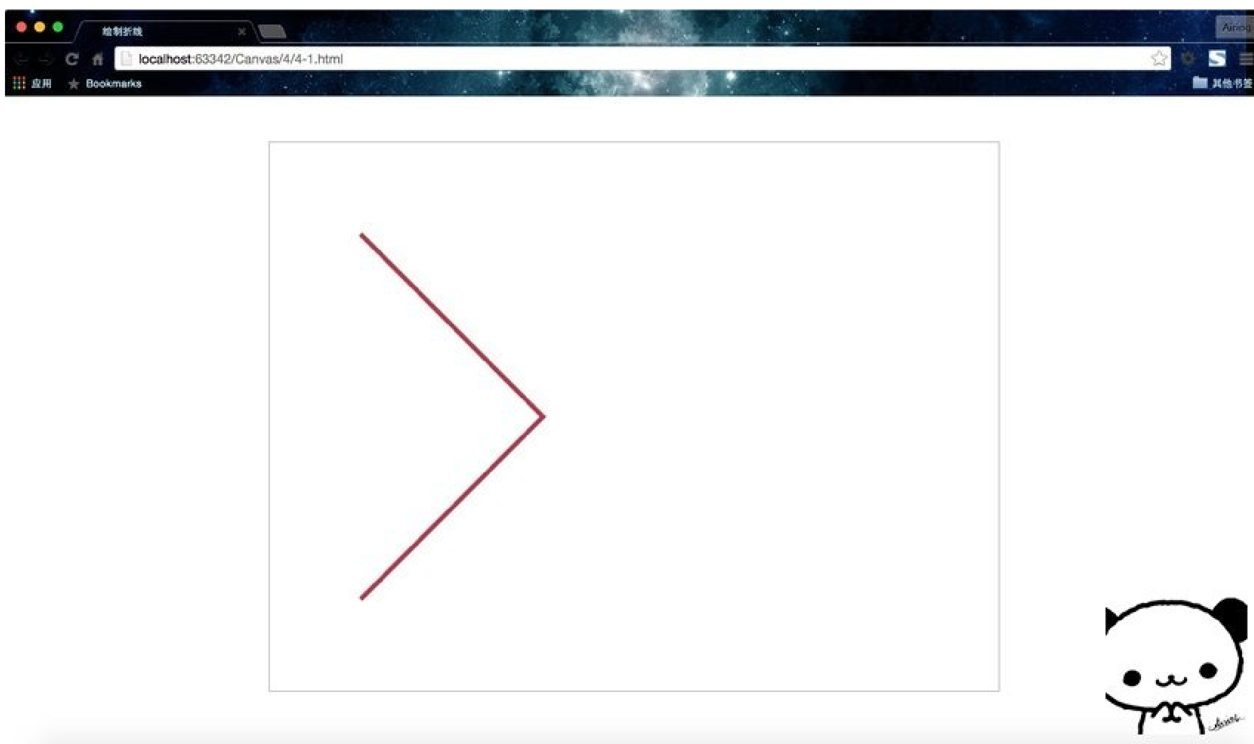
```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制折线</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.moveTo(100,100);
        context.lineTo(300,300);
        context.lineTo(100,500);
        context.lineWidth = 5;
        context.strokeStyle = "#AA394C";
        context.stroke();
    }
</script>
</body>
</html>
```

演示 4-1

运行结果：



绘制多条折线

那同理，我们要绘制多条样式各不相同的折线怎么办呢？比如我们在这里画三条折线，分别是红色、蓝色、黑色。聪明的小伙伴肯定想到了，这还不简单，只需要平移一下再改下画笔颜色就行了。代码格式都一样的，复制就可以了。代码如下。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制折线</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; display: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
```

```
            var canvas = document.getElementById("canvas");
            canvas.width = 800;
            canvas.height = 600;
            var context = canvas.getContext("2d");

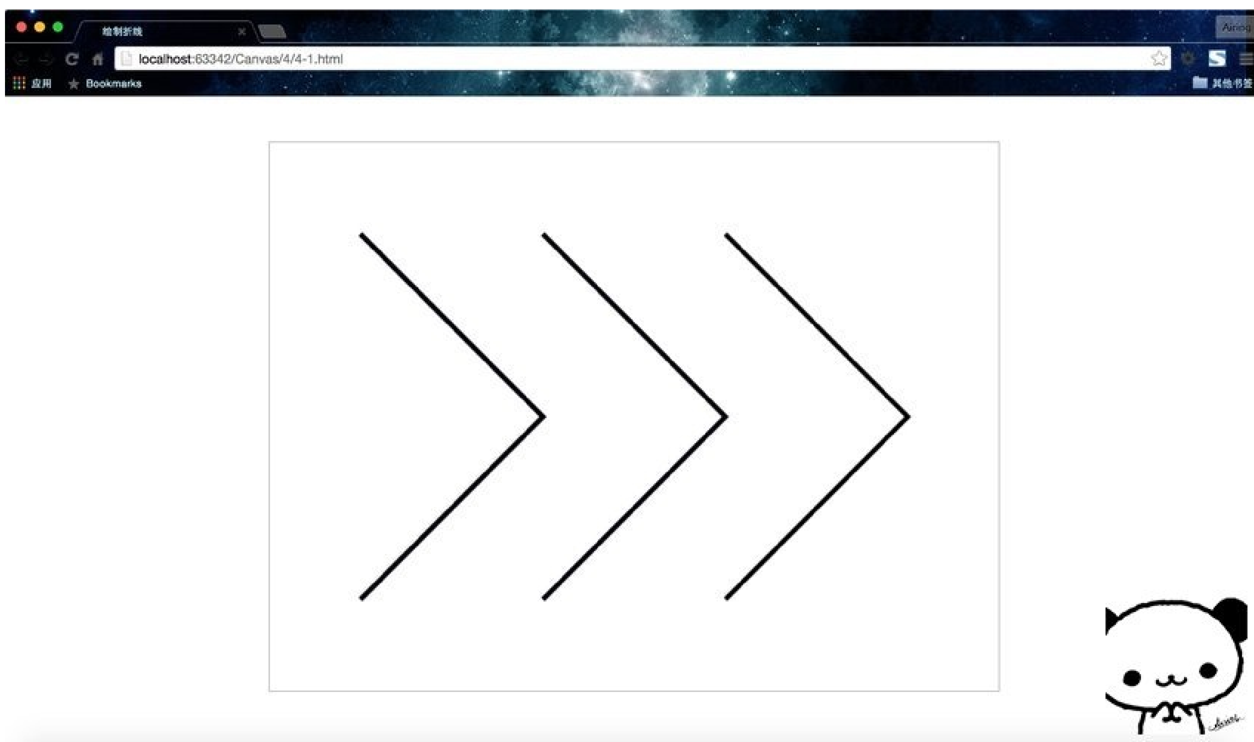
            context.moveTo(100,100);
            context.lineTo(300,300);
            context.lineTo(100,500);
            context.lineWidth = 5;
            context.strokeStyle = "red";
            context.stroke();

            context.moveTo(300,100);
            context.lineTo(500,300);
            context.lineTo(300,500);
            context.lineWidth = 5;
            context.strokeStyle = "blue";
            context.stroke();

            context.moveTo(500,100);
            context.lineTo(700,300);
            context.lineTo(500,500);
            context.lineWidth = 5;
            context.strokeStyle = "black";
            context.stroke();
        }
</script>
</body>
</html>
```

演示 4-2

运行结果：



咦？是不是很奇怪？说好的先红色，再蓝色，再黑色呢？怎么全是黑色了？其实，这里的原因是我之前一直强调的一点——**Canvas**是基于状态的绘制。

什么意思呢？其实这段代码每次使用 `stroke()` 时，它都会把之前设置的状态再绘制一遍。第一次 `stroke()` 时，绘制一条红色的折线；第二次 `stroke()` 时，会再重新绘制之前的那条红色的折线，但是这个时候的画笔已经被更换成蓝色的了，所以画出的折线全是蓝色的。换言之， `strokeStyle` 属性被覆盖了。同理，第三次绘制的时候，画笔颜色是最后的黑色，所以会重新绘制三条黑色的折线。所以，这里看到的三条折线，其实绘制了3次，一共绘制了6条折线。

那么，我想绘制三条折线，难道就没有办法了吗？艺术家之魂到此为止了么？没救了么？不，还有办法。

使用 beginPath() 开始绘制

为了让绘制方法不重复绘制，我们可以在每次绘制之前加上 `beginPath()` ，代表下次绘制的起始之处为 `beginPath()` 之后的代码。我们在三次绘制之前分别加上 `context.beginPath()` 。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
```

```
    <title>绘制折线</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.beginPath();
        context.moveTo(100,100);
        context.lineTo(300,300);
        context.lineTo(100,500);
        context.lineWidth = 5;
        context.strokeStyle = "red";
        context.stroke();

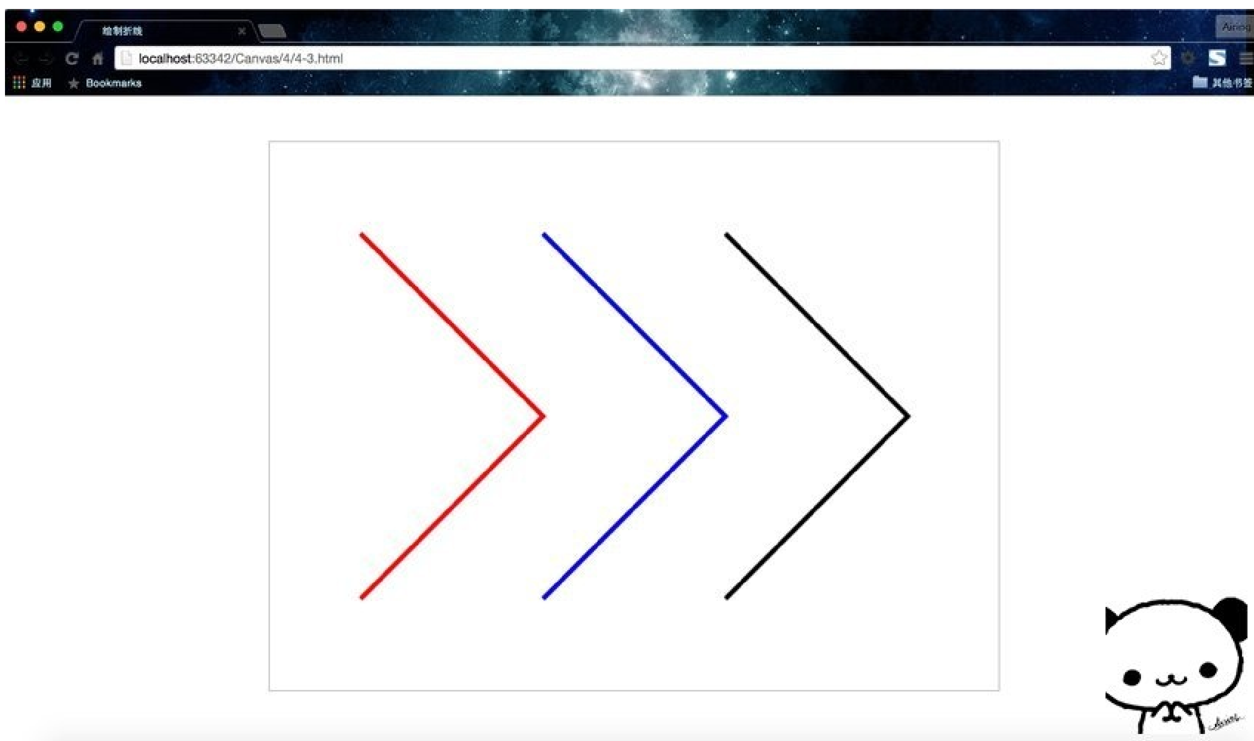
        context.beginPath();
        context.moveTo(300,100);
        context.lineTo(500,300);
        context.lineTo(300,500);
        context.lineWidth = 5;
        context.strokeStyle = "blue";
        context.stroke();

        context.beginPath();
        context.moveTo(500,100);
        context.lineTo(700,300);
        context.lineTo(500,500);
        context.lineWidth = 5;
        context.strokeStyle = "black";
        context.stroke();
```

```
    }
</script>
</body>
</html>
```

演示 4-3

运行结果：



可以看到，这里得到了我们预想的结果。因为使用了 `beginPath()` ，所以这里的绘制过程如我们所想的那样，只绘制了三次，而且每次只绘制一条折线。 `beginPath()` 是绘制设置状态的起始点，它之后代码设置的绘制状态的作用域结束于绘制方法 `stroke()` 、 `fill()` 或者 `closePath()` ，至于 `closePath()` 之后会讲到。

所以我们每次开始绘制前都务必要使用 `beginPath()` ，为了代码的完整性，建议大家在每次绘制结束后使用 `closePath()` 。大多数情况下不使用是没有什么关系的，但是使用的话可以增强代码的可读性以及意想不到的效果。什么效果呢？下一节我会介绍。

剧透一下，下一节我们开始绘制矩形！！怎么样，从上一节的“一画”（线条），到这一节的“两画”（折线），以及下一节课的“四画”（矩形），是不是很激动呢？让我们向艺术家之路继续前进！😋

Ch5 绘制矩形

使用closePath()闭合图形

首先我们用上节课的方法绘制一个矩形。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制矩形</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

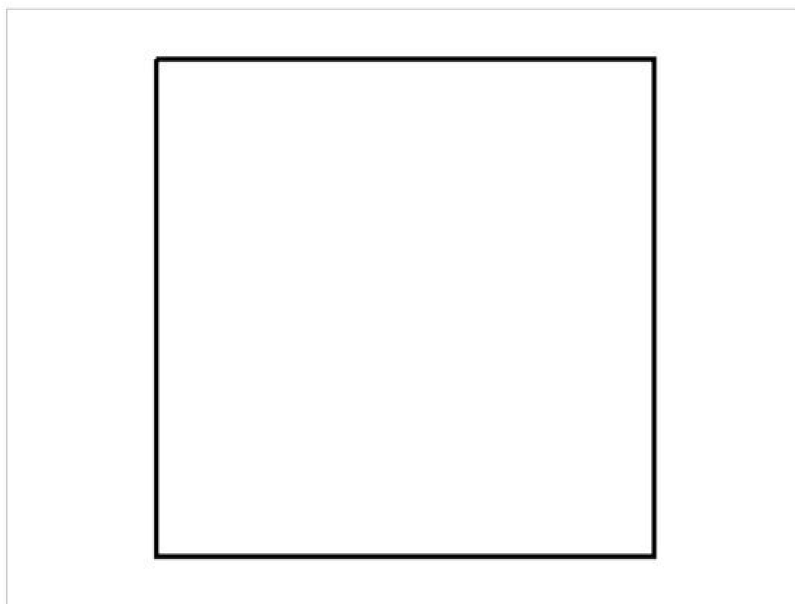
        context.beginPath();
        context.moveTo(150,50);
        context.lineTo(650,50);
```

```
            context.lineTo(650,550);
            context.lineTo(150,550);
            context.lineTo(150,50);      //绘制最后一笔使图像闭合
            context.lineWidth = 5;
            context.strokeStyle = "black";
            context.stroke();

        }
</script>
</body>
</html>
</body>
</html>
```

演示 5-1

运行结果：





乍一看没啥问题，但是视力好的童鞋已经发现了，最后一笔闭合的时候有问题，导致左上角有一个缺口。这种情况是设置了 `lineWidth` 导致的。如果默认1笔触的话，是没有问题的。但是笔触越大，线条越宽，这种缺口就越明显。那么这种情况该怎么避免呢？

标题已经剧透了，使用clothPath()闭合图形。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制矩形</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

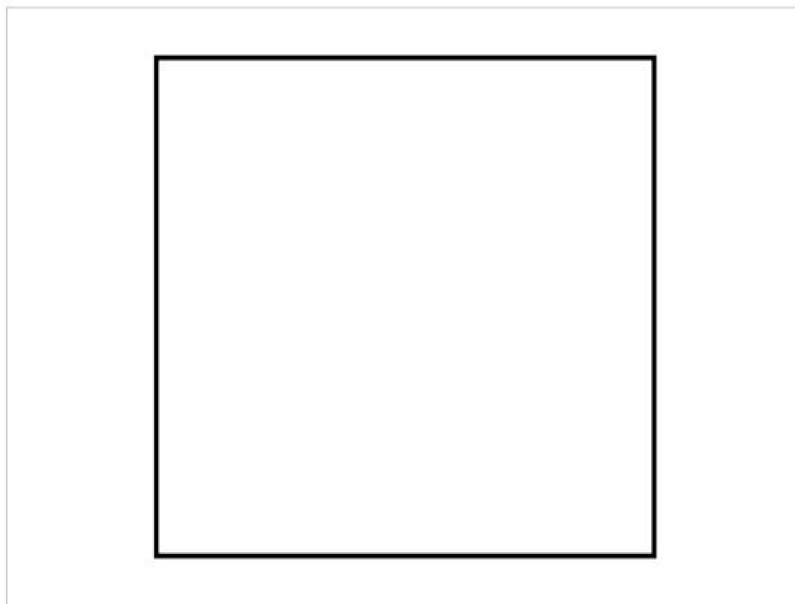
        context.beginPath();
        context.moveTo(150,50);
        context.lineTo(650,50);
        context.lineTo(650,550);
        context.lineTo(150,550);
        context.lineTo(150,50);     //最后一笔可以不画
        context.closePath();        //使用closePath()闭合图形

        context.lineWidth = 5;
        context.strokeStyle = "black";
        context.stroke();

    }
</script>
</body>
</html>
</body>
</html>
```

演示 5-2

运行结果：





这个例子结合上节课的讲解，我们知道了绘制路径的时候需要将规划的路径使用 `beginPath()` 与 `closePath()` 包裹起来。当然，最后一笔可以不画出来，直接使用 `closePath()` ，它会自动帮你闭合的。(所以如果你不想绘制闭合图形就不可以使用 `closePath()`)

给矩形上色

这里我们要介绍一个和 `stroke()` 同等重要的方法 `fill()` 。和当初描边一样，我们在填色之前，也要先用 `fillStyle` 属性选择要填充的颜色。

比如我们要给上面的矩形涂上黄色，可以这样写。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制矩形</title>
</head>
<body>
```

```
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.beginPath();
        context.moveTo(150,50);
        context.lineTo(650,50);
        context.lineTo(650,550);
        context.lineTo(150,550);
        context.lineTo(150,50);     //最后一笔可以不画
        context.closePath();        //使用closePath()闭合图形

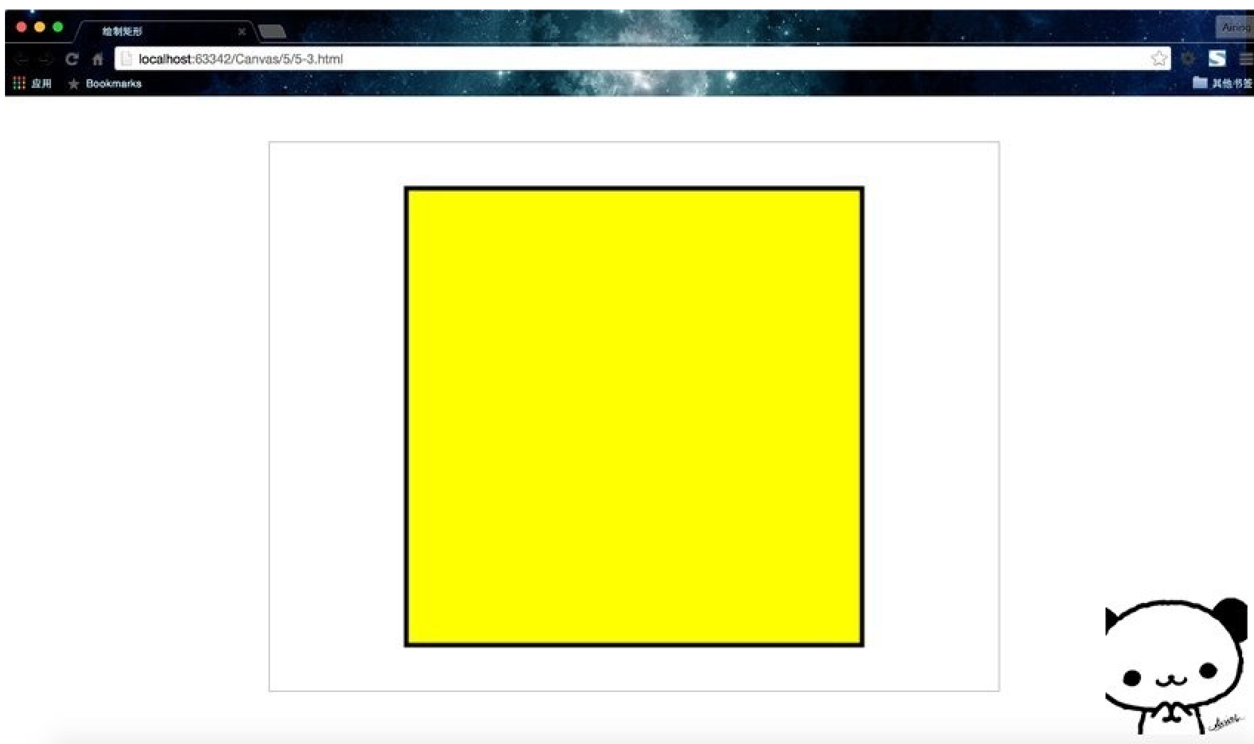
        context.fillStyle = "yellow";   //选择油漆桶的颜色
        context.lineWidth = 5;
        context.strokeStyle = "black";

        context.fill();                 //确定填充
        context.stroke();

    }
</script>
</body>
</html>
</body>
</html>
```

演示 5-3

运行结果：



这里需要注意的是一个良好的编码规范。因为之前说过了Canvas是基于状态的绘制，只有调用了 `stroke()` 和 `fill()` 才确定绘制。所以我们要把这两行代码放在最后，其余的状态设置的代码放在它们之前，将状态设置与确定绘制的代码分隔开。增强代码的可读性。

封装绘制方法

大家一定发现了，绘制矩形其实都是这样的四笔，我们每次用这种笨方法画矩形都要画这四笔有多麻烦，如果我们要花10个矩形？100个？1000个？都这样写，代码会臃肿，复用性很低。这里，我们可以使用JavaScript封装这些方法。我们知道，一个矩形可以由它的左上角坐标和其长宽唯一确定。

JavaScript函数

这里我们介绍一下JavaScript函数。如果有基础的同学可以跳过这一大段，直接看后面的。

JavaScript 和 ActionScript 语言的函数声明调用一样，都是编程语言中最简单的。

函数的作用

函数的作用，可以写一次代码，然后反复地重用这个代码。如:我们要完成多组数和的功能。

```
var sum;
sum = 3+2;
alert(sum);

sum=7+8 ;
alert(sum);

....  //不停重复两行代码
```

如果要实现8组数的和，就需要16行代码，实现的越多，代码行也就越多。所以我们可以把完成特定功能的代码块放到一个函数里，直接调用这个函数，就省去重复输入大量代码的麻烦。

使用函数完成:

```
function add2(a,b){
    sum = a + b;
     alert(sum);
} //  只需写一次就可以

add2(3,2);
add2(7,8);
....  //只需调用函数就可以
```

定义函数

如何定义一个函数呢？看看下面的格式：

```
function  函数名( )
{
     函数体;
}
```

function定义函数的关键字，“函数名”你为函数取的名字，“函数体”替换为完成特定功能的代码。

函数调用

函数定义好后，是不能自动执行的，需要调用它，直接在需要的位置写函数名。一般有两种方式：

- 第一种情况：在 `<script>` 标签内调用。

```
<script>
function tcon()
{
    alert("恭喜你学会函数调用了!");
}
tcon();    //调用函数，直接写函数名。
</script>
```

- 第二种情况：在HTML文件中调用，如通过点击按钮后调用定义好的函数。

这种情况以后用到了再说。

有参数的函数

格式如下：

```
function 函数名(参数1,参数2)
{
     函数代码
}
```

注意:参数可以多个，根据需要增减参数个数。参数之间用(逗号，) 隔开。

按照这个格式，函数实现任意两个数的和应该写成：

```
function add2(x,y)
{
    sum = x + y;
    document.write(sum);
}
```

x和y则是函数的两个参数，调用函数的时候，我们可通过这两个参数把两个实际的加数传递给函数了。

例如，add2(3，4)会求3+4的和，add2(60,20)则会求出60和20的和。

返回值函数

```
function add2(x,y)
{
    sum = x + y;
    return sum; //返回函数值,return后面的值叫做返回值。
}
```

这里的return和其他语言中的一样，但是在JS或者AS等弱类型语言中，返回值类型是不需要写到函数声明里的。

至此，我们把JavaScript函数也顺便系统的说了一下。下面回到正题~

我们也可以封装一下我们的矩形，代码如下：

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>封装绘制矩形方法</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
```

```
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        drawRect(context, 150, 50, 50, 50, "red", 5, "blue");
        drawRect(context, 250, 50, 50, 50, "green", 5, "red");
        drawRect(context, 350, 50, 50, 50, "yellow", 5, "black")
;
    }

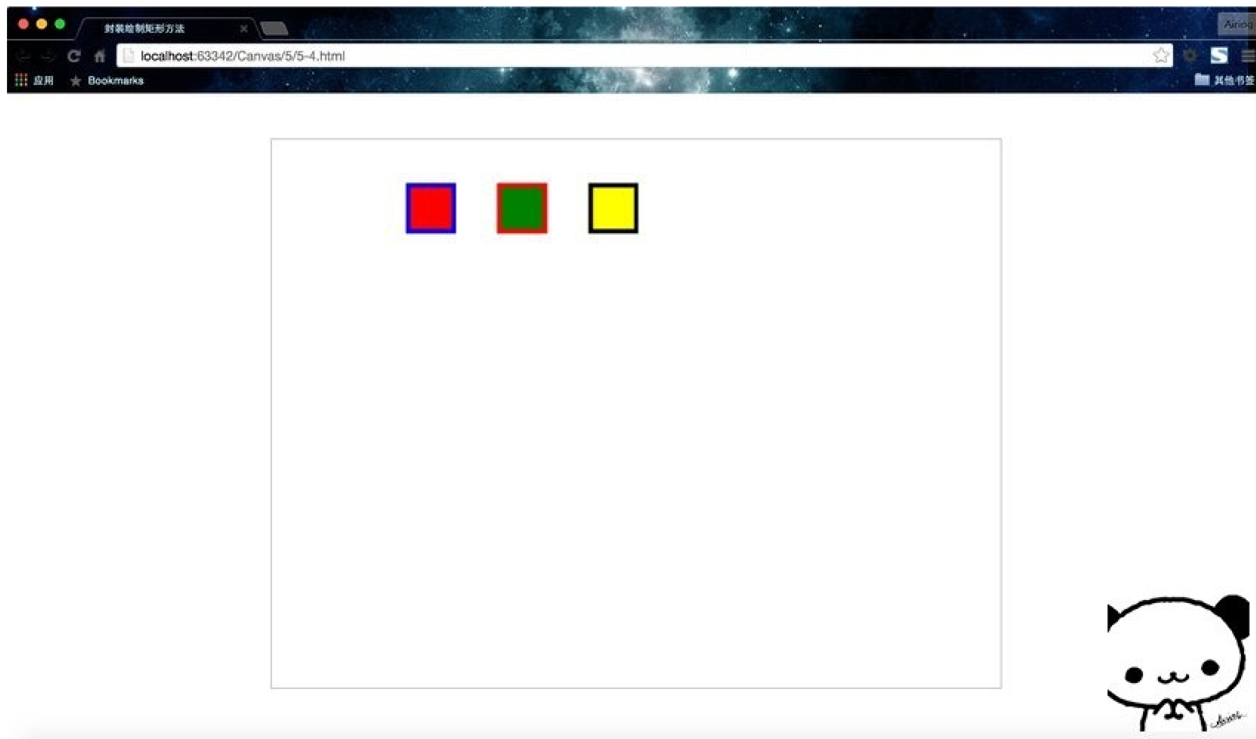
    function drawRect(cxt, x, y, width, height, fillColor, borde
rWidth, borderColor){
        cxt.beginPath();
        cxt.moveTo(x, y);
        cxt.lineTo(x + width, y);
        cxt.lineTo(x + width, y + height);
        cxt.lineTo(x, y + height);
        cxt.lineTo(x, y);
        cxt.closePath();

        cxt.lineWidth = borderWidth;
        cxt.strokeStyle = borderColor;
        cxt.fillStyle = fillColor;

        cxt.fill();
        cxt.stroke();
    }
</script>
</body>
</html>
</body>
</html>
```

演示 5-4

运行结果：



使用rect()方法绘制矩形

犹豫绘制矩形是常用的方法，所以在Canvas API中已经帮我们封装好了一个绘制矩形的方法—— `rect()` 。这个方法接收4个参数x, y, width, height，实际调用时也就是

```
context.rect(x,y,width,height);
```

基于此，我们帮刚才封装的方法优化一下。

```
function drawRect(cxt, x, y, width, height, fillColor, borderWid
th, borderColor){
        cxt.beginPath();
        cxt.rect(x, y, width, height);
        //cxt.closePath();    可以不用closePath()

        cxt.lineWidth = borderWidth;
        cxt.strokeStyle = borderColor;
        cxt.fillStyle = fillColor;

        cxt.fill();
        cxt.stroke();
}
```

调用封装方法，绘制魔性图像

还记得我们第三节开头那个魔性的图像吗？



好，我们就拿它开刀，练练手，给咱刚封装好的方法活动活动筋骨。

```
<!DOCTYPE html>
<html lang="zh">
<head>
```

```
    <meta charset="UTF-8">
    <title>绘制魔性图形</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.beginPath();
        context.rect(0, 0, 800, 600);
        context.fillStyle = "#AA9033";

        context.fill();

        context.beginPath();
        for(var i=0; i<=20; i++){
            drawWhiteRect(context, 200 + 10 * i, 100 + 10 * i, 4
00 - 20 * i, 400 - 20 * i);
            drawBlackRect(context, 205 + 10 * i, 105 + 10 * i, 3
90 - 20 * i, 390 - 20 * i);
        }

        context.beginPath();
        context.rect(395, 295, 5, 5);
        context.fillStyle = "black";
        context.lineWidth = 5;

        context.fill();
        context.stroke();
    }
```

```
    function drawBlackRect(cxt, x, y, width, height){
        cxt.beginPath();
        cxt.rect(x, y, width, height);

        cxt.lineWidth = 5;
        cxt.strokeStyle = "black";

        cxt.stroke();
    }

    function drawWhiteRect(cxt, x, y, width, height){
        cxt.beginPath();
        cxt.rect(x, y, width, height);

        cxt.lineWidth = 5;
        cxt.strokeStyle = "white";

        cxt.stroke();
    }
</script>
</body>
</html>
</body>
</html>
```

演示 5-5

运行结果：



是不是很有魔性？是不是非常的酷？这段代码就不花篇幅解释了，大家自己课下琢磨琢磨，也可以尝试着用已经学过的知识去绘制一个酷酷的图像。这节课内容有点多，其实也只是介绍了四个属性和方法——`closePath()`、`fillStyle`、`fill()`、`rect()`，还有一个扩展的JavaScript函数讲解。

好了，多花点时间消化消化。然后我们带着我们完成的艺术品，继续前进~😋

Ch6 线条的属性

线条属性概述

线条的属性共有以下四个：

1、lineCap属性

lineCap 定义上下文中线的端点，可以有以下 3 个值。

- butt：默认值，端点是垂直于线段边缘的平直边缘。
- round：端点是在线段边缘处以线宽为直径的半圆。
- square：端点是在选段边缘处以线宽为长、以一半线宽为宽的矩形。

2、lineJoin属性

lineJoin 定义两条线相交产生的拐角，可将其称为连接。在连接处创建一个填充三角形，可以使用 lineJoin 设置它的基本属性。

- miter：默认值，在连接处边缘延长相接。miterLimit 是角长和线宽所允许的最大比例(默认是 10)。
- bevel：连接处是一个对角线斜角。
- round：连接处是一个圆。

3、线宽

lineWidth 定义线的宽度(默认值为 1.0)。

4、笔触样式

strokeStyle 定义线和形状边框的颜色和样式。

后面两个前面已经说过了，这里我们着重来看看前两个属性。

线条的帽子 `lineCap`

废话不多说，直接上代码看效果。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>线条的帽子</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.lineWidth = 50;
        context.strokeStyle = "#1BAAAA";

        context.beginPath();
        context.moveTo(100,100);
        context.lineTo(700,100);
        context.lineCap = "butt";
        context.stroke();

        context.beginPath();
        context.moveTo(100,300);
        context.lineTo(700,300);
        context.lineCap = "round";
        context.stroke();

        context.beginPath();
        context.moveTo(100,500);
```

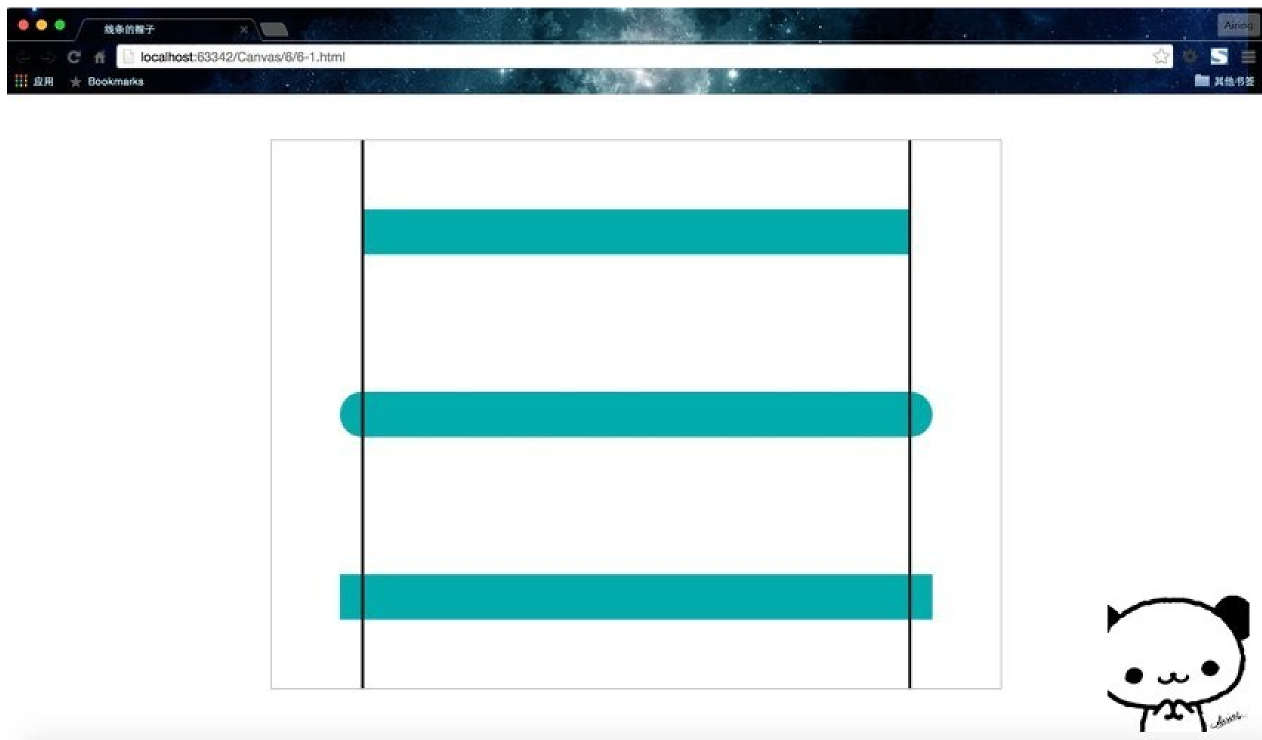
```
        context.lineTo(700,500);
        context.lineCap = "square";
        context.stroke();

        //下面画两个基准线方便观察
        context.lineWidth = 3;
        context.strokeStyle = "black";

        context.beginPath();
        context.moveTo(100,0);
        context.lineTo(100,600);
        context.moveTo(700,0);
        context.lineTo(700,600);
        context.stroke();
    }
</script>
</body>
</html>
```

演示 6-1

运行结果：



这里我还做了两条平行线做一下参考，这样一眼就能看清 `lineCap` 三个值的特点。但要注意，这个帽子只在线条的端点处起作用，哪怕是折点很多的折线，也仅仅是在开始和终止的两个端点带帽子。如果想改变线条折点(两个线段的连接处)的样式，那就要用到下面的 `lineJoin` 属性。

线条的连接 `lineJoin`

废话不多说，直接上代码看效果。这段代码改自4-3，只是设置了一下连接的属性。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>线条的连接</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; display: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.beginPath();
        context.moveTo(100,100);
        context.lineTo(300,300);
        context.lineTo(100,500);
        context.lineJoin = "miter";
        context.lineWidth = 20;
        context.strokeStyle = "red";
```

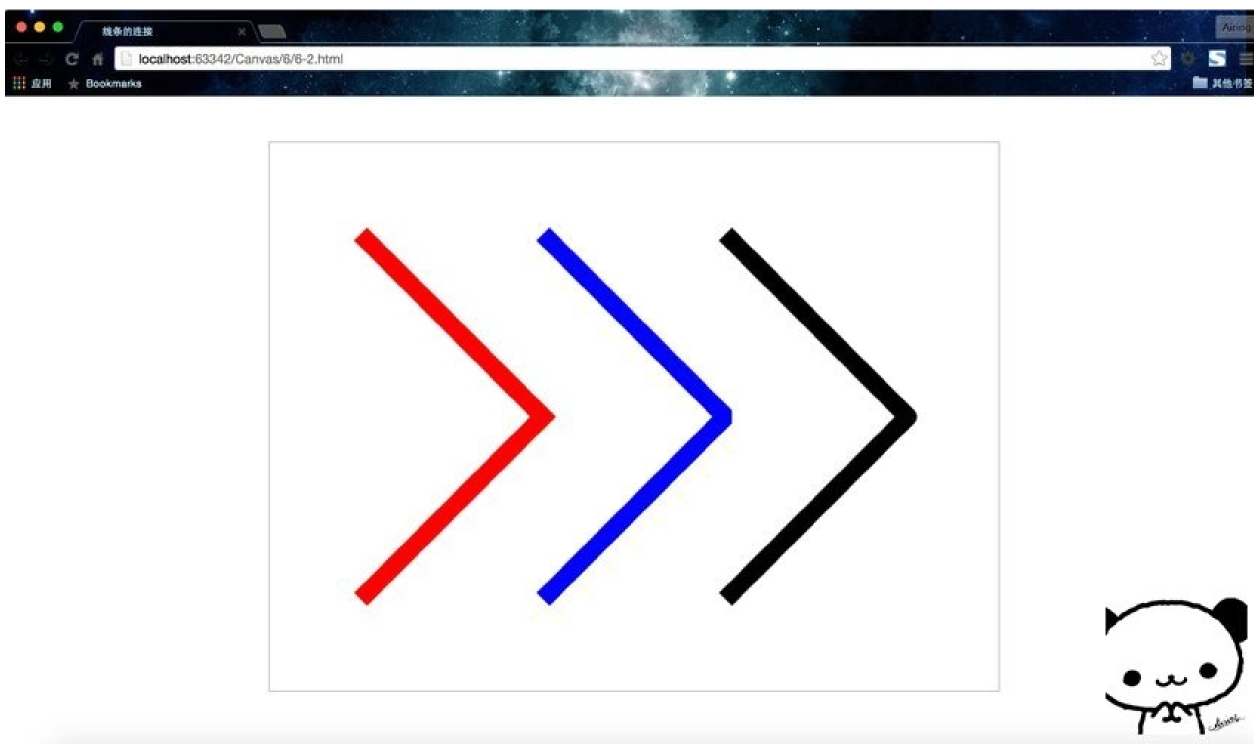
```
        context.stroke();

        context.beginPath();
        context.moveTo(300,100);
        context.lineTo(500,300);
        context.lineTo(300,500);
        context.lineJoin = "bevel";
        context.lineWidth = 20;
        context.strokeStyle = "blue";
        context.stroke();

        context.beginPath();
        context.moveTo(500,100);
        context.lineTo(700,300);
        context.lineTo(500,500);
        context.lineJoin = "round";
        context.lineWidth = 20;
        context.strokeStyle = "black";
        context.stroke();
    }
</script>
</body>
</html>
```

演示 6-2

运行结果：



看不清的童鞋自己放大网页或者自己将代码的线宽调宽一点。

这里有一个很隐蔽的属性，就是当 `lineJoin` 设置为miter时（默认），会解锁一个成绩，可以使用 `miterLimit` 属性。

举个例子看看。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>miterLimit</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
```

```
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

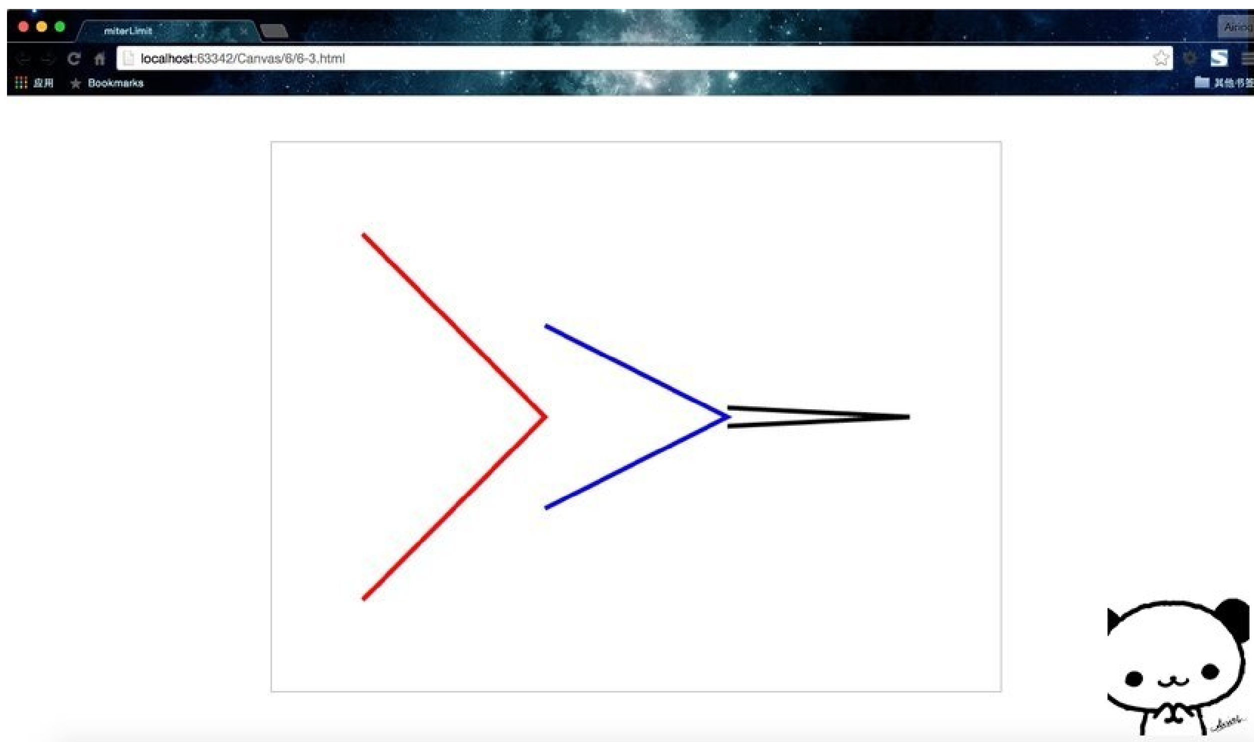
        context.beginPath();
        context.moveTo(100,100);
        context.lineTo(300,300);
        context.lineTo(100,500);
        context.lineJoin = "miter";
        context.miterLimit = 10;
        context.lineWidth = 5;
        context.strokeStyle = "red";
        context.stroke();

        context.beginPath();
        context.moveTo(300,200);
        context.lineTo(500,300);
        context.lineTo(300,400);
        context.lineJoin = "miter";
        context.miterLimit = 10;
        context.lineWidth = 5;
        context.strokeStyle = "blue";
        context.stroke();

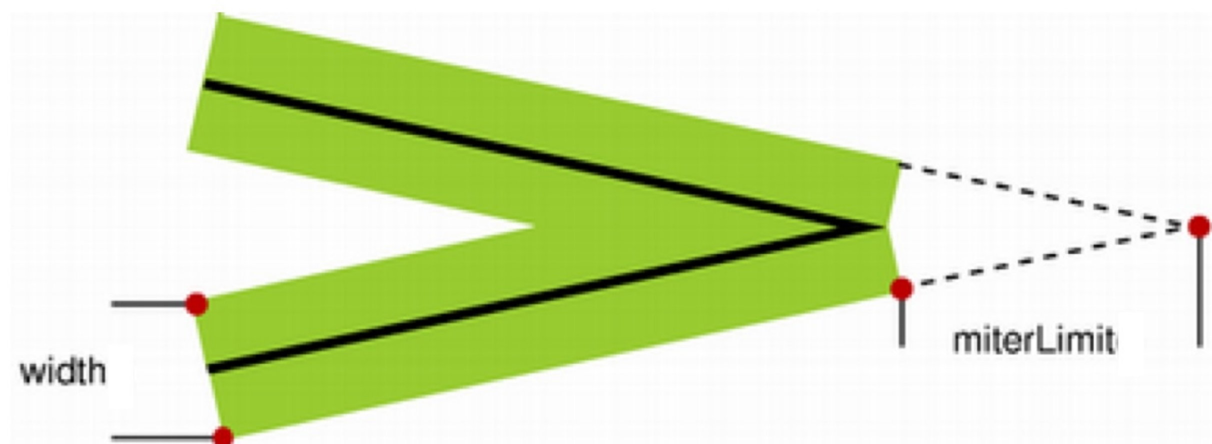
        context.beginPath();
        context.moveTo(500,290);
        context.lineTo(700,300);
        context.lineTo(500,310);
        context.lineJoin = "miter";
        context.miterLimit = 10;
        context.lineWidth = 5;
        context.strokeStyle = "black";
        context.stroke();
    }
</script>
</body>
</html>
```

演示 6-3

运行结果：



会发现，这个 `miterLimit` 规定了一个自动填充连接点的极限值。如果超过了这个值，会导致 `lineJoin` 属性失效，会从 miter 变成 bevel。可以看出来，这个值和线宽以及夹角有关，具体是啥关系。看下图。



可以看到，关系有点复杂。有兴趣的小伙伴可以推导一下这个值与线宽、夹角的函数关系。其实，大多时候用不到这个隐藏属性，即便用到了也是凭感觉写个数然后不满意再调试即可。

高级线段绘制举例

实际在画布上绘制线段时，会有一些奇怪的现象发生，这里融合本节课学到的两个线段的属性讲解一个。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>miterLimit</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        // 实例1: 圆形端点,斜角连接,在画布左上角
        context.beginPath();
        context.moveTo(0,0);
        context.lineTo(180,0);
        context.lineTo(180,180);
        context.lineJoin = 'bevel';
        context.lineCap = 'round';
        context.lineWidth = 10;
        context.strokeStyle = "red";
        context.stroke();

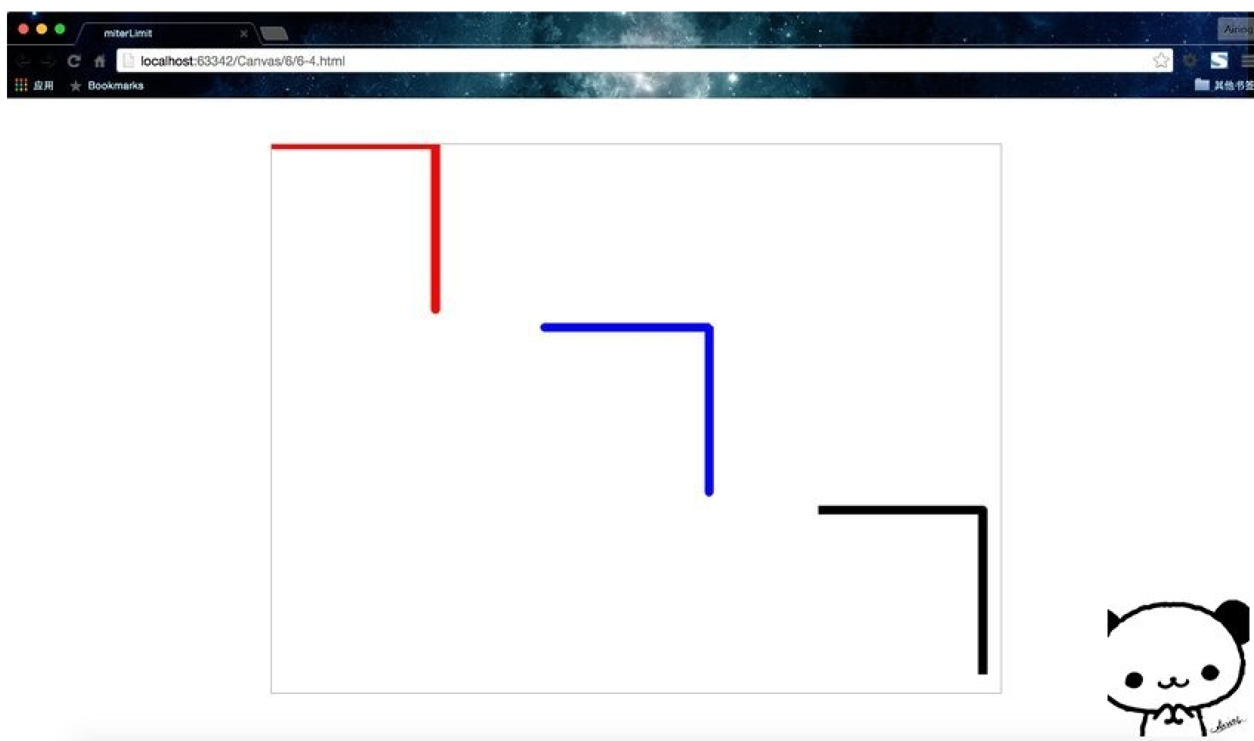
        // 实例2: 圆形端点,斜角连接,不在画布左上角
        context.beginPath();
        context.moveTo(300,200);
        context.lineTo(480,200);
        context.lineTo(480,380);
```

```
        context.lineJoin = 'bevel';
        context.lineCap = 'round';
        context.lineWidth = 10;
        context.strokeStyle = "blue";
        context.stroke();

        // 实例3: 平直端点,圆形连接,不在画布左上角
        context.beginPath();
        context.moveTo(600,400);
        context.lineTo(780,400);
        context.lineTo(780,580);
        context.lineJoin = 'round';
        context.lineCap = 'butt';
        context.lineWidth = 10;
        context.strokeStyle = "black";
        context.stroke();
    }
</script>
</body>
</html>
```

演示 6-4

运行结果：



这 3 个线段和连接的实例有助于说明在画布上绘制线段时不同属性的组合。

实例 1 尝试从画布左上角开始绘制，结果发生了一个奇怪的现象。Canvas 路径在 x 轴和 y 轴方向上都绘制到了起点的外侧。由于这个原因实例 1 上面的线看起来要细些。另外，左上角水平部分中圆形端点也无法看到，它们都被绘制到了屏幕之外的负值坐标区域。此外，`lineJoin` 定义的对角线斜角也没有绘出。

实例 2 调整了例子 1 中出现的问题,将起始点离开左上角。这样就绘制出了完整的水平线,并且圆形 `lineCap` 和斜角 `lineJoin` 都被绘制出来了。

实例 3 显示了去掉 `lineCap` 设置后的默认端点效果,并且将 `lineJoin` 调整为圆角。

至此，线条的所有内容我们已经说完了。这个看起来很简单的一个线条，原来还有这么多内容！艺术家可不是那么好当的，不过想必也激发了童鞋们的求知欲。Canvas究竟有多少内容等待着我们去发现？这是一个开始，让我们继续前行！😋

Ch7 填充颜色

艺术离不开色彩，今天咱们来介绍一下填充颜色，体会一下色彩的魅力。

填充颜色主要分为两种：

1. 基本颜色
2. 渐变颜色（又分为线性渐变与径向渐变）

我们一个个来看。

填充基本颜色

Canvas `fillStyle` 属性用来设置画布上形状的基本颜色和填充。`fillStyle` 使用简单的颜色名称。这看起来非常简单，例如:

```
context.fillStyle = "red";
```

下面是出自 HTML4 规范的可用颜色字符串值列表，共十六个。由于 HTML5 没有修改专属的颜色，HTML4 的颜色都可以在 HTML5 中正确显示。

名称	名称字符串	十六进制数字字符串
黑色	Black	#000000s
绿色	Green	#008000
银色	Silver	#C0C0C0
石灰色	Lime	#00FF00
灰色	Gray	#808080
橄榄色	Olive	#808000
白色	White	#FFFFFF
黄色	Yellow	#FFFF00
栗色	Maroon	#800000
海蓝色	Navy	#000080
红色	Red	#FF0000
蓝色	Blue	#0000FF
紫色	Purple	#800080
深蓝绿色	Teal	#008080
紫红色	Fuchsia	#FF00FF
浅蓝绿色	Aqua	#00FFFF

所有这些颜色值都可以应用到 `strokeStyle` 属性和 `fillStyle` 属性中。

好了，我来总结一下填充基本色的方法：（也可用于 `strokeStyle` 属性）

（1） 使用颜色字符串填充。

```
context.fillStyle = "red";
```

（2）使用十六进制数字字符串填充。

```
context.fillStyle = "#FF0000";
```

（3）使用十六进制数字字符串简写形式填充。

```
context.fillStyle = "#F00";
```

（4）使用 `rgb()` 方法设置颜色。

```
context.fillStyle = "rgb(255,0,0)";
```

（5）使用 `rgba()` 方法设置颜色。

```
context.fillStyle = "rgba(255,0,0,1)";
```

此方法最后一个参数传递的是alpha值，透明度范围为1（不透明）~0（透明）。

（6）使用 `hsl()` 方法设置颜色。

```
context.fillStyle = "hsl(0,100%,50%)";
```

HSL即是代表色相（H），饱和度（S），明度（L）三个通道的颜色。

（7）使用 `hsla()` 方法设置颜色。

```
context.fillStyle = "hsla(0,100%,50%,1)";
```

以上7句代码都是填充"#FF0000"这个红色。

填充渐变形状

在画布上创建渐变填充有两个基本选项：线性或径向。线性渐变创建一个水平、垂直或者对角线的填充图案。径向渐变自中心点创建一个放射状填充。填充渐变形状分为三步：添加渐变线，为渐变线添加关键色，应用渐变。下面是它们的一些示例。

线性渐变

三步走战略：

1. 添加渐变线：

```
var grd = context.createLinearGradient(xstart,ystart,xend,ye
nd);
```

2. 为渐变线添加关键色(类似于颜色断点)：

```
grd.addColorStop(stop,color);
```

这里的stop传递的是 0 ~ 1 的浮点数，代表断点到(xstart,ystart)的距离占整个渐变色长度是比例。

1. 应用渐变：

```
context.fillStyle = grd;
context.strokeStyle = grd;
```

写个代码来看看。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>填充线性渐变</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
```

```
        var context = canvas.getContext("2d");

        context.rect(200,100,400,400);

        //添加渐变线
        var grd = context.createLinearGradient(200,300,600,300);

        //添加颜色断点
        grd.addColorStop(0,"black");
        grd.addColorStop(0.5,"white");
        grd.addColorStop(1,"black");

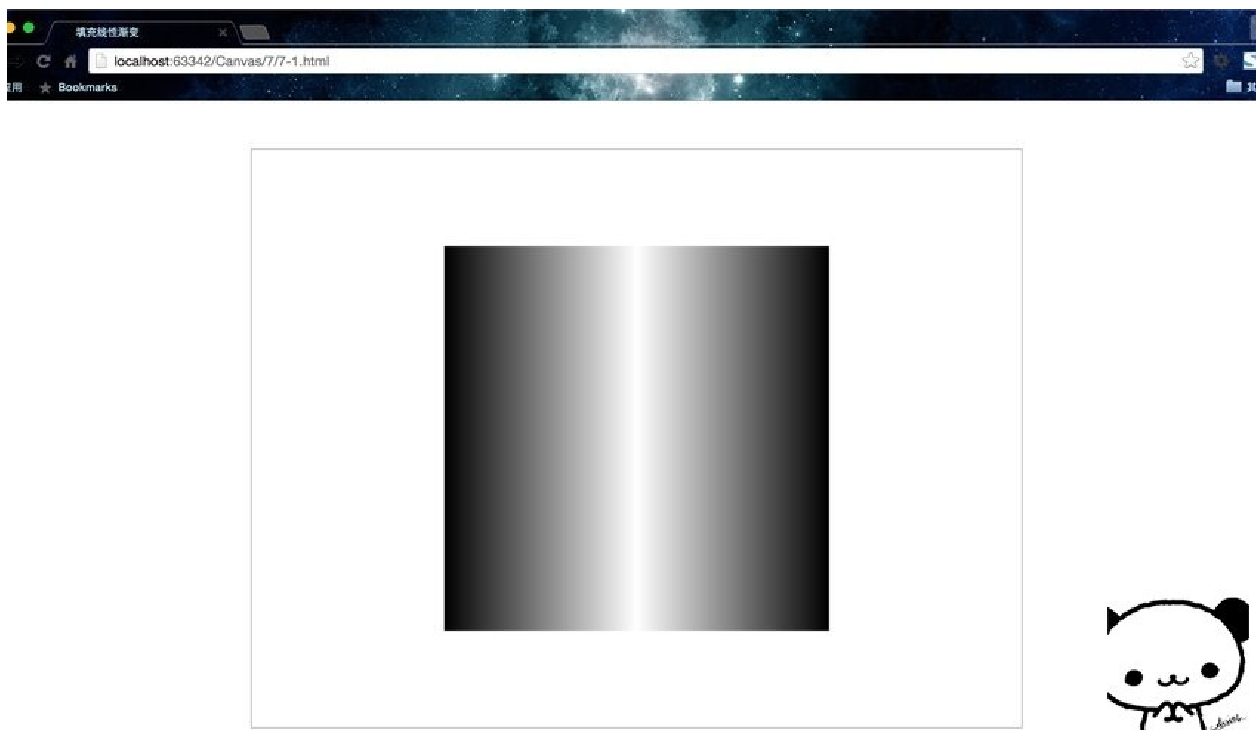
        //应用渐变
        context.fillStyle = grd;

        context.fill();

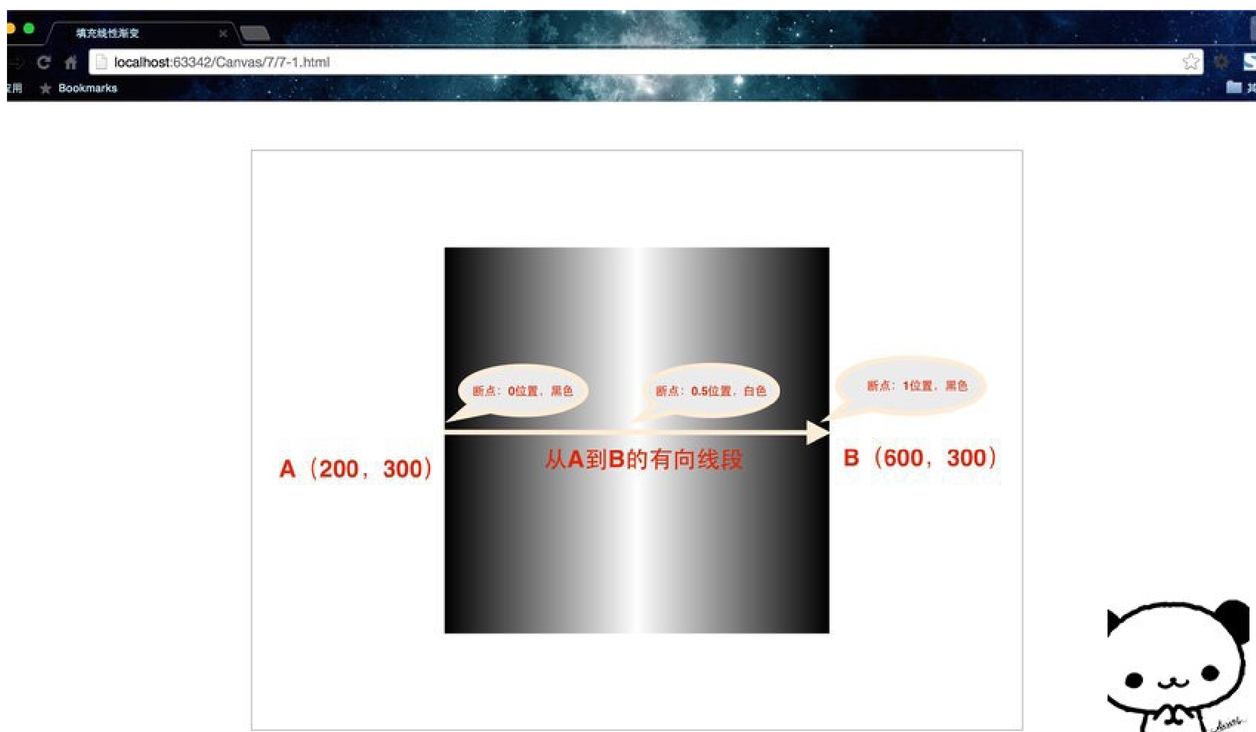
    }
</script>
</body>
</html>
```

演示 7-1

运行结果：



我觉得有必要做一个图解，方便大家一次性理解渐变。



为了方便理解，建议把渐变线看成是一个有向线段。如果熟悉PS等绘图工具，用过其中的渐变色设置，应该会很好理解。

这里渐变线的起点和终点不一定要在图像内，颜色断点的位置也是一样的。但是如果图像的范围大于渐变线，那么在渐变线范围之外，就会自动填充离端点最近的断点的颜色。

这里配合两个补充函数再举一例。

绘制矩形的快捷方法

`fillRect(x,y,width,height)`、`stroke(x,y,width,height)`。这两个函数可以分别看做 `rect()` 与 `fill()` 以及 `rect()` 与 `stroke()` 的组合。因为 `rect()` 仅仅只是规划路径而已，而这两个方法确实实实在在的绘制。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>填充线性渐变</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; display: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        //添加渐变线
        var grd = context.createLinearGradient(100,300,700,300);

        //添加颜色断点
        grd.addColorStop(0,"olive");
        grd.addColorStop(0.25,"maroon");
        grd.addColorStop(0.5,"aqua");
        grd.addColorStop(0.75,"fuchsia");
        grd.addColorStop(0.25,"teal");
```

```
        //应用渐变
        context.fillStyle = grd;
        context.strokeStyle = grd;

        context.strokeRect(200,50,300,50);
        context.strokeRect(200,100,150,50);
        context.strokeRect(200,150,450,50);

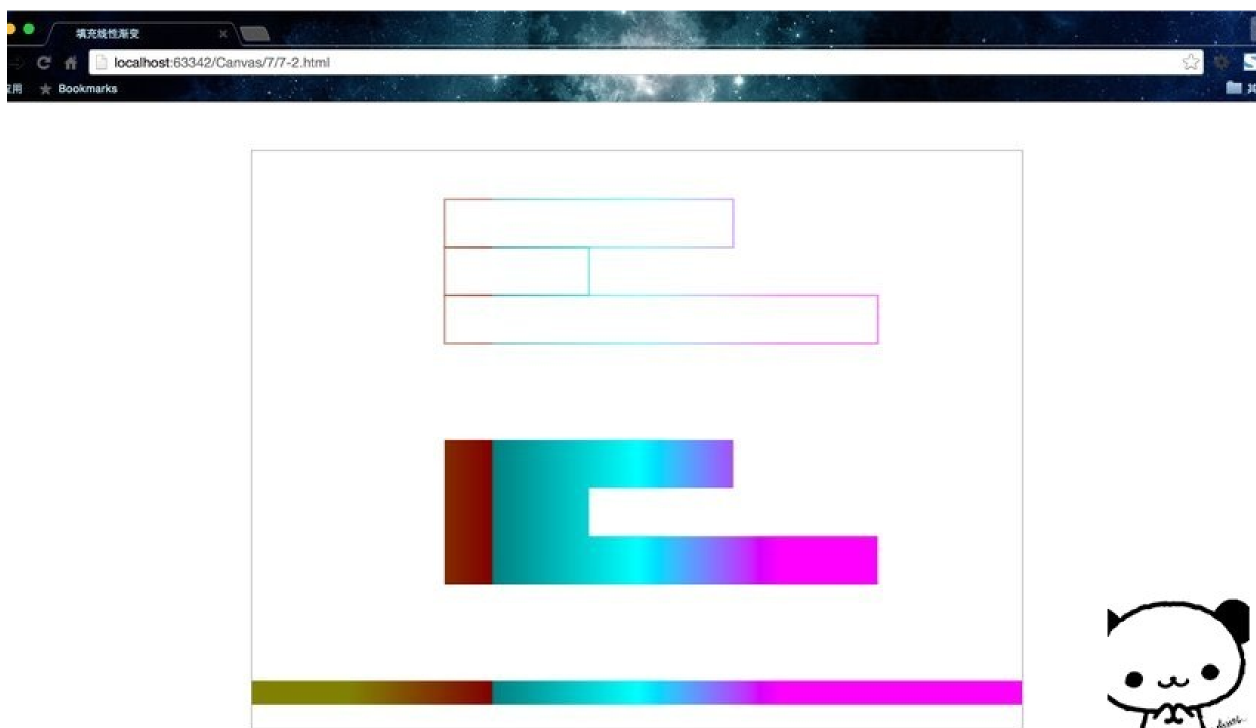
        context.fillRect(200,300,300,50);
        context.fillRect(200,350,150,50);
        context.fillRect(200,400,450,50);

        context.fillRect(0,550,800,25);

    }
</script>
</body>
</html>
```

演示 7-2

运行结果：



这两个页面都是水平渐变，但是要清楚线性渐变不一定是水平的，方向可以是任意的，通过渐变线的端点来设置方向。

径向渐变

同样是三步走战略，只不过是第一步的所用方法变了。

1. 添加渐变圆：

```
var grd = context.createRadialGradient(x0,y0,r0,x1,y1,r1);
```

2. 为渐变线添加关键色(类似于颜色断点)：

```
grd.addColorStop(stop,color);
```

3. 应用渐变：

```
context.fillStyle = grd;
context.strokeStyle = grd;
```

线性渐变是基于两个端点定义的，但是径向渐变是基于两个圆定义的。

我们把示例7-2改写一下。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>填充径向渐变</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; display: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>
```

```
<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        //添加渐变线
        var grd = context.createRadialGradient(400,300,100,400,3
00,200);

        //添加颜色断点
        grd.addColorStop(0,"olive");
        grd.addColorStop(0.25,"maroon");
        grd.addColorStop(0.5,"aqua");
        grd.addColorStop(0.75,"fuchsia");
        grd.addColorStop(0.25,"teal");

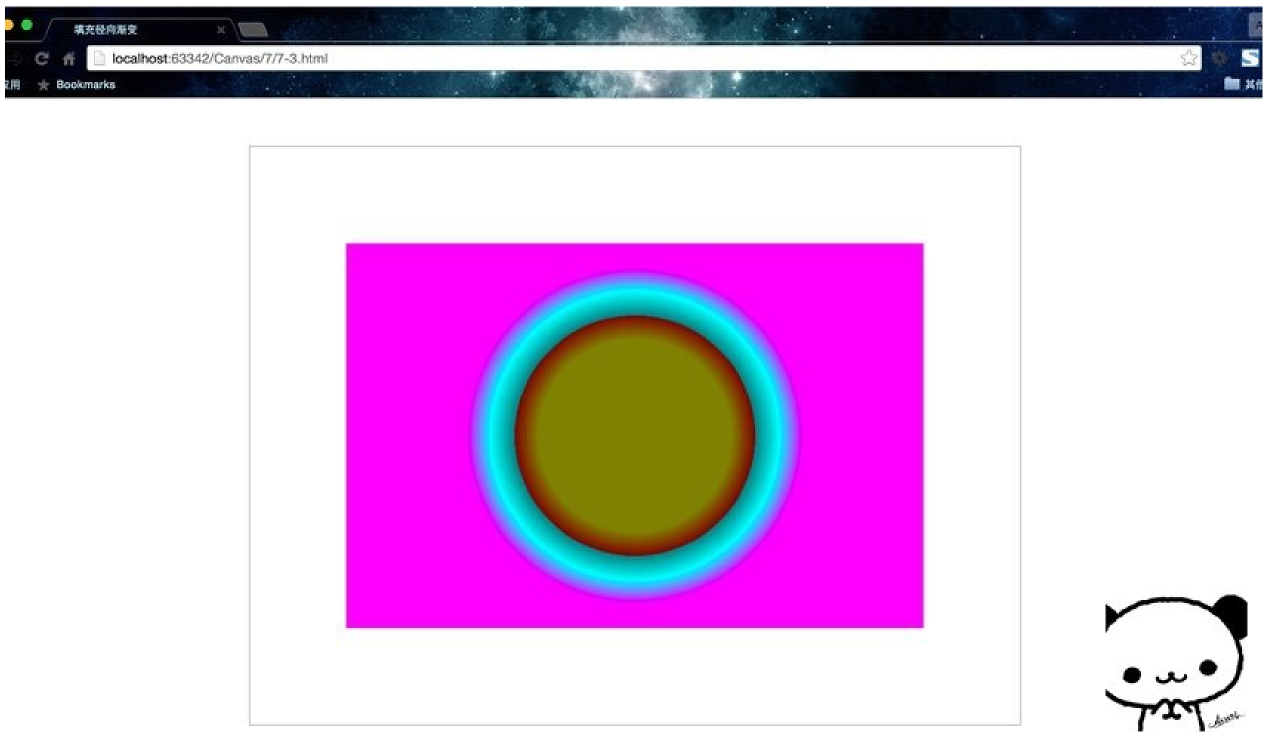
        //应用渐变
        context.fillStyle = grd;

        context.fillRect(100,100,600,400);


    }
</script>
</body>
</html>
```

演示 7-3

运行结果：



怎么感觉这个颜色搭配那么的......算了，这个就叫做艺术。

`createRadialGradient(x0,y0,r0,x1,y1,r1);` 方法规定了径向渐变开始和结束的范围，即两圆之间的渐变。

总结一下，这节课我们学习了 `fillStyle` 、 `createLinearGradient()` 、 `createRadialGradient()` 、 `addColorStop()` 、 `fillRect()` 、 `strokeRect()` 等属性和方法，详细介绍了填充基本色、线性渐变、径向渐变。

好了，现在学会了上色，那么尽情的使用色彩，绘制出属于我们自己的艺术品吧！



Ch8 填充样式

`createPattern()` 简介

纹理其实就是图案的重复，填充图案通过 `createPattern()` 函数进行初始化。它需要传进两个参数 `createPattern(img,repeat-style)` ，第一个是Image对象实例，第二个参数是String类型，表示在形状中如何显示repeat图案。可以使用这个函数加载图像或者整个画布作为形状的填充图案。

有以下4种图像填充类型：

- 平面上重复：repeat;
- x轴上重复：repeat-x;
- y轴上重复：repeat-y;
- 不使用重复：no-repeat;

其实 `createPattern()` 的第一个参数还可以传入一个canvas对象或者video对象，这里我们只讲解Image对象，其余的大家自己尝试。

创建并填充图案

首先看一下怎么加载图像：

1. 创建Image对象
2. 为Image对象指定图片源

代码如下：

```
var img = new Image();     //创建Image对象
img.src = "8-1.jpg";     //为Image对象指定图片源
```

> 扩展：HTML中的相对路径
>
> './目录或文件名' 或者 '目录或文件名' 是指当前操作的文件所在目录的路径
>
> '../目录或文件名' 是指当前所操作的文件所在目录的上一级目录的路径

之后填充纹理：

```
var pattern = context.createPattern(img,"repeat");
context.fillStyle = pattern;
```

我们直接看一段完整的程序，这里我要重复填充这个萌萌的长颈鹿作为纹理。需要注意的是，选择图片时一定要选择那种左右互通，上下互通的图片做为纹理，这样看上去才不会有不自然的短接处。



安利一个网站。这张图取自优美图网，这个网站非常赞，里面的图片非常漂亮而且种类繁多，最重要的是它是免费的！！！我从初中开始就一直使用这个网站来找素材。并且最近又推出了APP，有Android和iOS端，推荐大家下载使用。（这是个秘密，我一般不告诉别人的。）

下面提供代码。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>填充纹理</title>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas" style="border: 1px solid #aaaaaa; displa
y: block; margin: 50px auto;">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        var img = new Image();
        img.src = "8-1.jpg";
        img.onload = function(){
            var pattern = context.createPattern(img, "repeat");
            context.fillStyle = pattern;
            context.fillRect(0,0,800,600);
        }

    }
</script>
</body>
</html>
```

演示 8-1

运行结果：





这里使用了 `Image` 的 `onload` 事件，它的作用是对图片进行预加载处理，即在图片加载完成后才立即除非其后function的代码体。这个是必须的，如果不写的话，画布将会显示黑屏。因为没有等待图片加载完成就填充纹理，导致浏览器找不到图片。

这里使用了"repeat"，童鞋们也可尝试使用一下其他三个值，看看会有什么不同的效果。也可以自己找一下其他的图片尝试填充，看看效果。

还有，长颈鹿是不是特别萌？看来我们已经是个艺术家了呢！这节内容很少，我们继续前进！😋

Ch9 绘制标准圆弧

高级路径

今天开始，我们就要征战路径最后也是最难的部分了——高级路径。之前我们学习的都是绘制线条（基本路径），接下来的四节课我们详细看看绘制曲线（高级路径）的有关方法。

剧透一下，主要有四个方法：

- 标准圆弧：`arc()`
- 复杂圆弧：`arcTo()`
- 二次贝塞尔曲线：`quadraticCurveTo()`
- 三次贝塞尔曲线：`bezierCurveTo()`

在开始之前，我们优化一下我们的作图环境。灵感来自于上节课的纹理，如果不喜欢这个背景，我在images目录下还提供了其他的背景图，供大家选择。另外把所有的样式表都写在了 `<head>` 下。

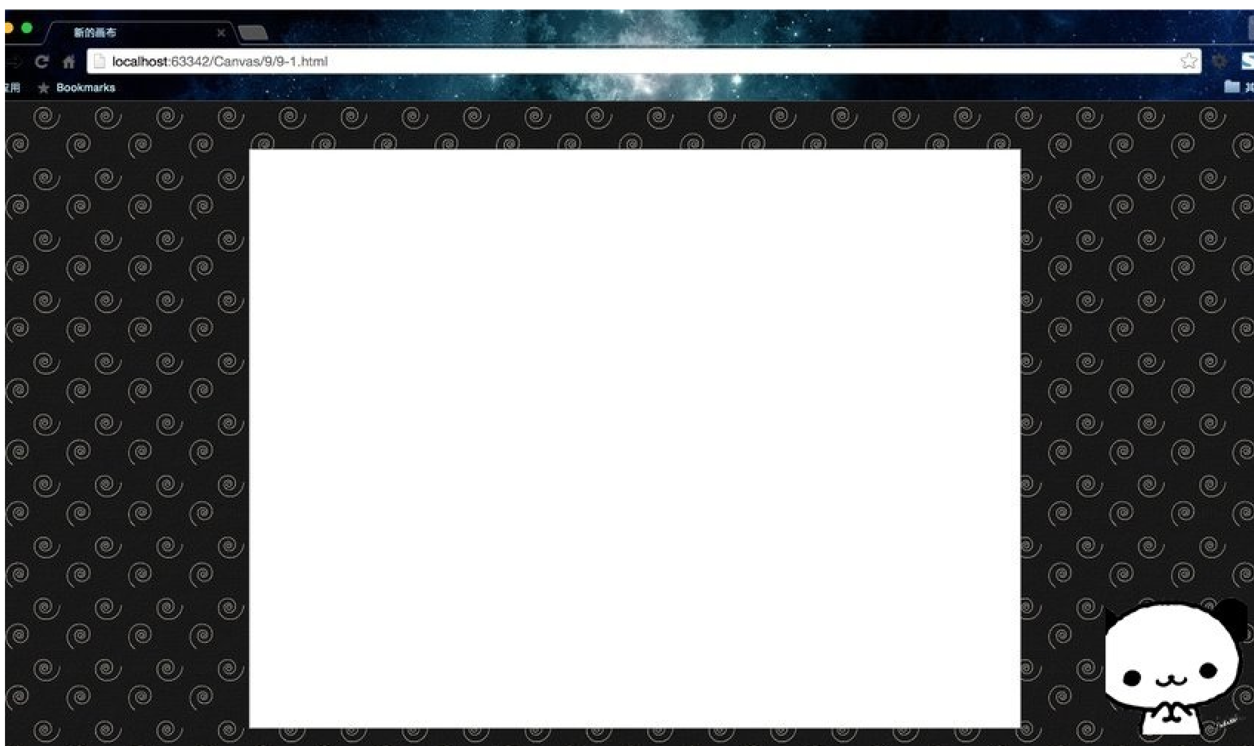
```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>新的画布</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

    }
</script>
</body>
</html>
```

演示 9-1

运行结果：



之所以要绘制一个空白的矩形填满画布，是因为我们之前说过，canvas是透明的，如果不设置背景色，那么它就会被我设置的 `<body>` 纹理所覆盖，想要使其拥有背景色（白色），只有绘制矩形覆盖canvas这一个方法。

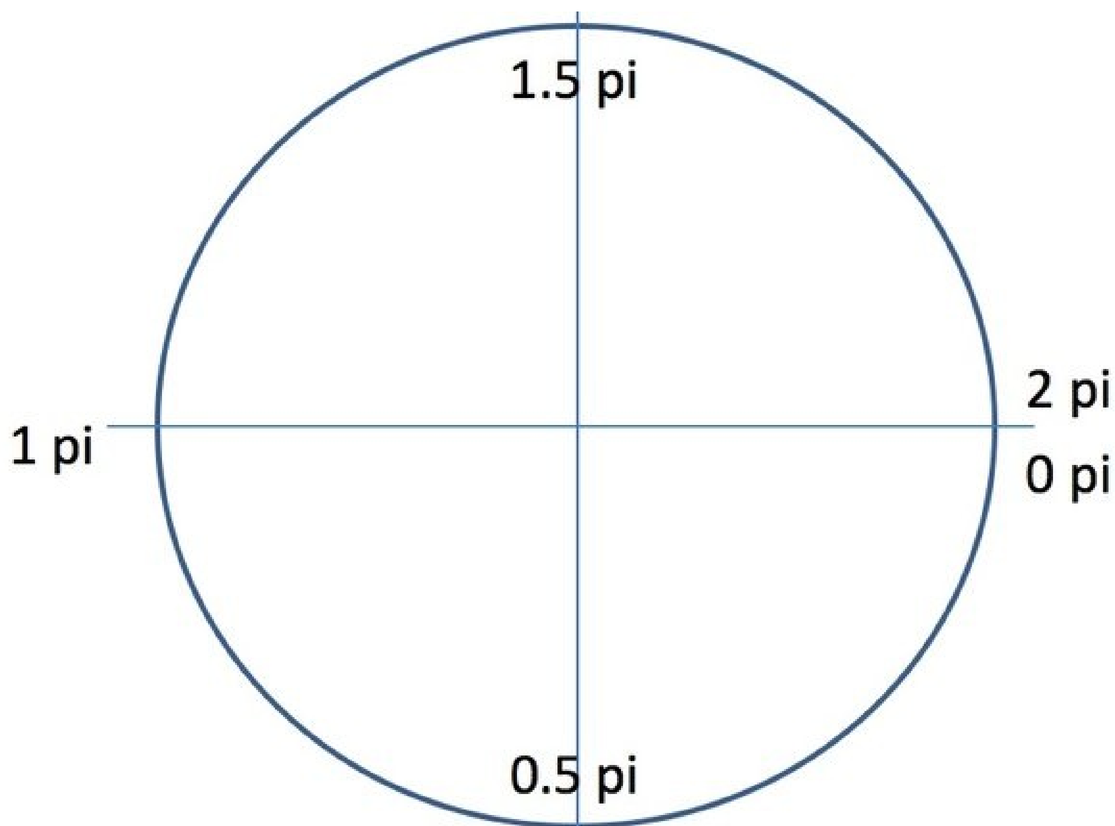
怎么样，是不是非常的酷？

使用 `arc()` 绘制圆弧

`arc()` 的使用方法如下：

`context.arc(x,y,radius,startAngle,endAngle,anticlockwise)`

前面三个参数，分别是圆心坐标与圆半径。 **`startAngle`** 、 **`endAngle`** 使用的是弧度值，不是角度值。弧度的规定是绝对的，如下图。



`anticlockwise` 表示绘制的方法，是顺时针还是逆时针绘制。它传入布尔值，true表示逆时针绘制，false表示顺时针绘制，缺省值为false。

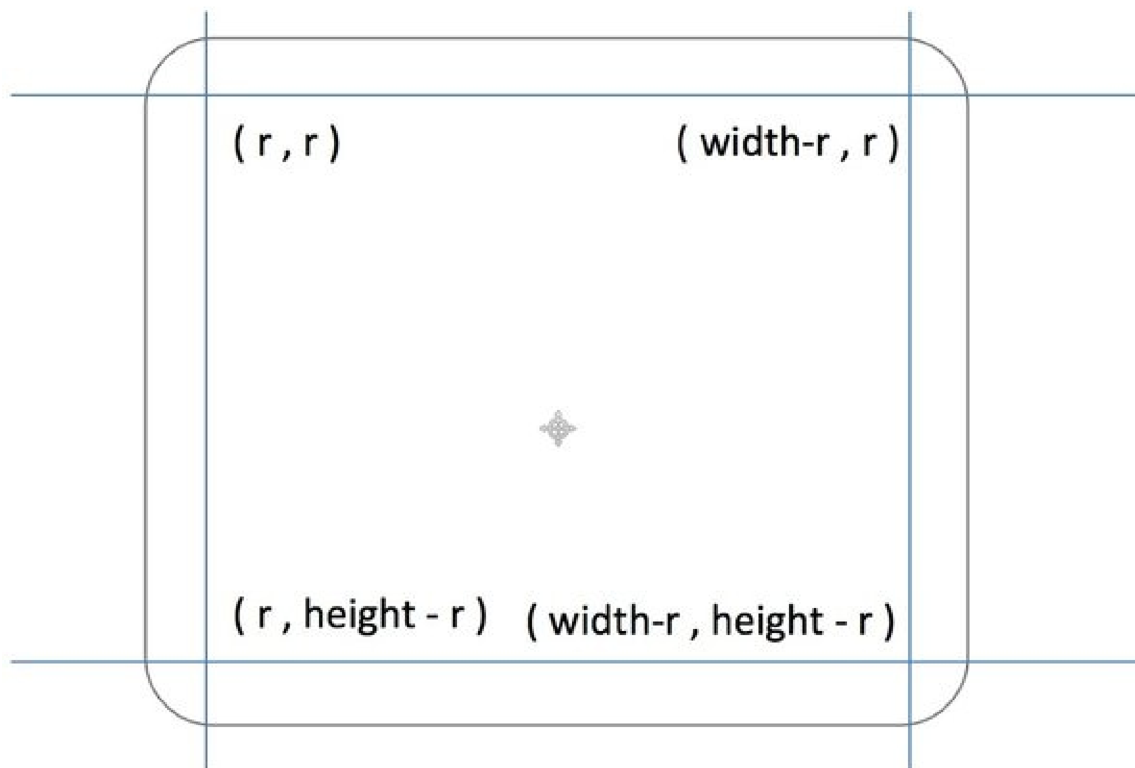
弧度的规定是绝对的，什么意思呢？就是指你要绘制什么样的圆弧，弧度直接按上面的那个标准填就行了。

比如你绘制 0.5pi ~ 1pi 的圆弧，如果顺时针画，就只是左下角那1/4个圆弧；如果逆时针画，就是与之互补的右上角的3/4圆弧。此处自己尝试，不再举例。

绘制圆角矩形

下面，我们结合基本路径和高级路径的知识，绘制一个圆角矩形。

圆角矩形是由四段线条和四个1/4圆弧组成，拆解如下。



因为我们要写的是函数而不是一个固定的圆角矩形，所以这里列出的是函数需要的参数。分析好之后，直接敲出代码。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>圆角矩形</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
```

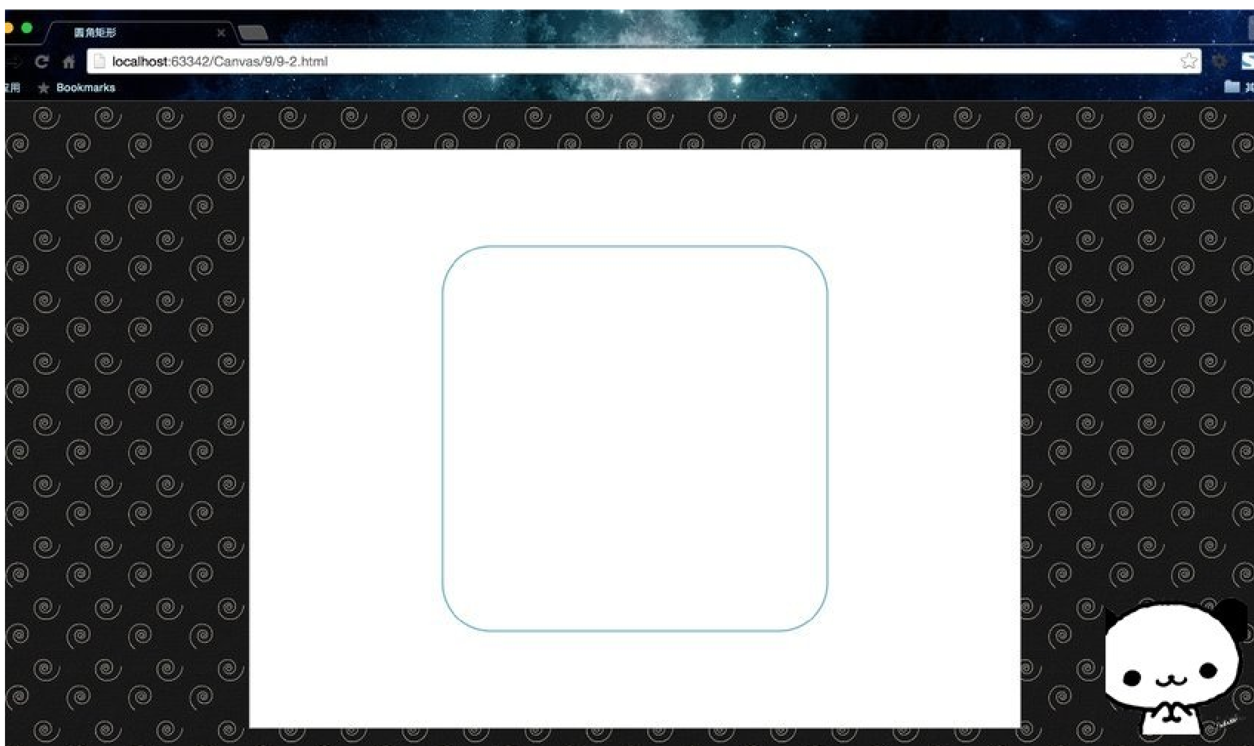
```
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        drawRoundRect(context, 200, 100, 400, 400, 50);
        context.strokeStyle = "#0078AA";
        context.stroke();
    }

    function drawRoundRect(cxt, x, y, width, height, radius){
        cxt.beginPath();
        cxt.arc(x + radius, y + radius, radius, Math.PI, Math.PI
 * 3 / 2);
        cxt.lineTo(width - radius + x, y);
        cxt.arc(width - radius + x, radius + y, radius, Math.PI
* 3 / 2, Math.PI * 2);
        cxt.lineTo(width + x, height + y - radius);
        cxt.arc(width - radius + x, height - radius + y, radius,
0, Math.PI * 1 / 2);
        cxt.lineTo(radius + x, height +y);
        cxt.arc(radius + x, height - radius + y, radius, Math.PI
 * 1 / 2, Math.PI);
        cxt.closePath();
    }
</script>
</body>
</html>
```

演示 9-2

运行结果：



建议大家自己动手绘制一个圆角矩形，这样有助于对路径的掌握。

下面我们用这个函数来做点其他的事情。

绘制2048游戏界面

对代码不做过多讲解，大家自己研究研究，建议自己动手先尝试写一下。因为我这里采用的是硬编码，所以不是很好，大家也可尝试优化一下。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>2048游戏界面</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
```

```
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        drawRoundRect(context, 200, 100, 400, 400, 5);
        context.fillStyle = "#AA7B41";
        context.strokeStyle = "#0078AA";
        context.stroke();
        context.fill();

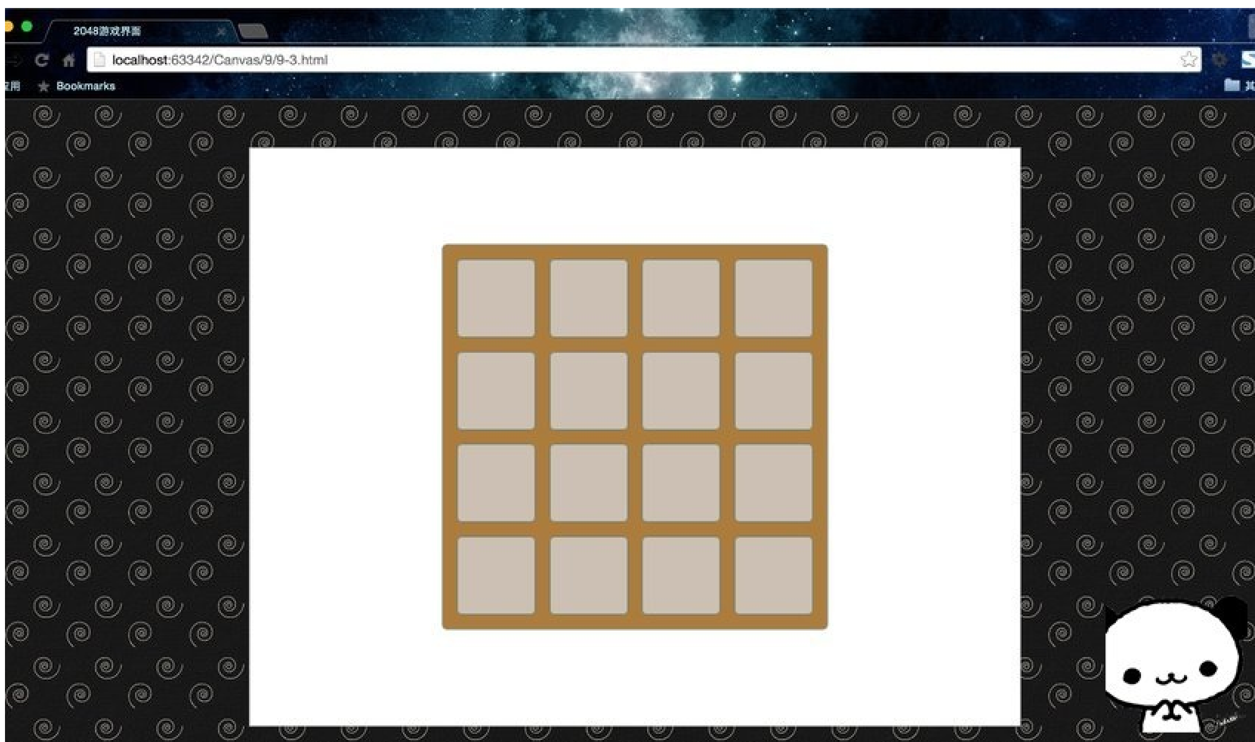
        for(var i = 1; i <= 4; i++){
            for(var j = 1; j <= 4; j++){
                drawRoundRect(context, 200 + 16 * i + 80 * (i - 1
), 100 + 16 * j + 80 * (j - 1), 80, 80, 5);
                context.fillStyle = "#CCBFB4";
                context.strokeStyle = "#0078AA";
                context.stroke();
                context.fill();
            }
        }
    }

    function drawRoundRect(cxt, x, y, width, height, radius){
        cxt.beginPath();
        cxt.arc(x + radius, y + radius, radius, Math.PI, Math.PI
 * 3 / 2);
        cxt.lineTo(width - radius + x, y);
        cxt.arc(width - radius + x, radius + y, radius, Math.PI
* 3 / 2, Math.PI * 2);
        cxt.lineTo(width + x, height + y - radius);
        cxt.arc(width - radius + x, height - radius + y, radius,
```

```
0, Math.PI * 1 / 2);
        cxt.lineTo(radius + x, height +y);
        cxt.arc(radius + x, height - radius + y, radius, Math.PI
 * 1 / 2, Math.PI);
        cxt.closePath();
    }
</script>
</body>
</html>
```

演示 9-3

运行结果：



这个圆角矩形的函数写好之后，可以自己封装进JS文件里，以后遇到什么好的函数都可以放进去，这样积累下来，这个文件就是一套属于自己的图形库和游戏引擎了，是不是非常的酷？

其实游戏制作是Canvas的主要用途，但是要知道每一个游戏设计师都是一个艺术家。是不是觉得前途一片光明？让我们继续前进！

Ch10 使用切点绘制圆弧

`arcTo()` 介绍

`arcTo()` 方法接收5个参数，分别是两个切点的坐标和圆弧半径。这个方法是依据切线画弧线，即由两个切线确定一条弧线。 具体如下。

`arcTo(x1,y1,x2,y2,radius)`

这个函数以给定的半径绘制一条弧线，圆弧的起点与当前路径的位置到(x1, y1)点的直线相切，圆弧的终点与(x1, y1)点到(x2, y2)的直线相切。因此其通常配合 `moveTo()` 或 `lineTo()` 使用。其能力是可以被更为简单的 `arc()` 替代的，其复杂就复杂在绘制方法上使用了切点。

使用切点绘制弧线

下面的案例我把切线也绘制出来了，看的更清楚一些。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制弧线</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>
```

```
<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        drawArcTo(context, 200, 200, 600, 200, 600, 400, 100);
    };

    function drawArcTo(cxt, x0, y0, x1, y1, x2, y2, r){
        cxt.beginPath();
        cxt.moveTo(x0, y0);
        cxt.arcTo(x1, y1, x2, y2, r);

        cxt.lineWidth = 6;
        cxt.strokeStyle = "red";
        cxt.stroke();

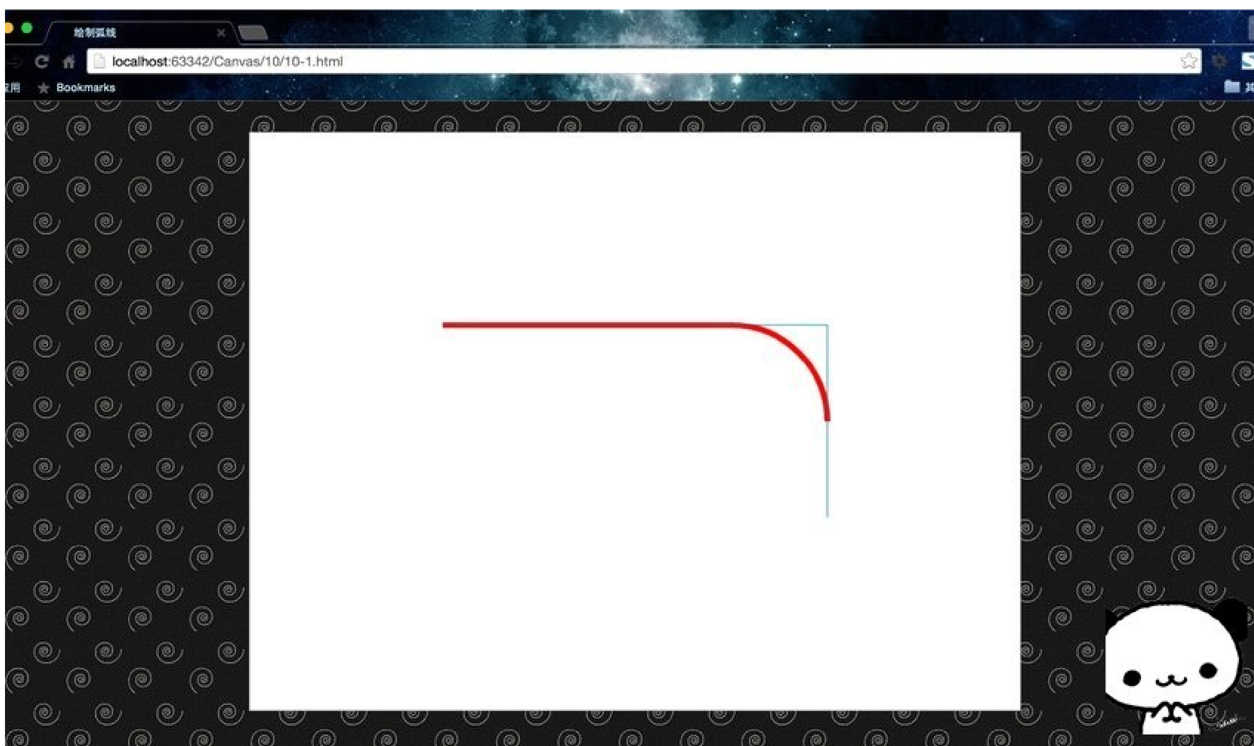
        cxt.beginPath();
        cxt.moveTo(x0, y0);
        cxt.lineTo(x1, y1);
        cxt.lineTo(x2, y2);

        cxt.lineWidth = 1;
        cxt.strokeStyle = "#0088AA";
        cxt.stroke();

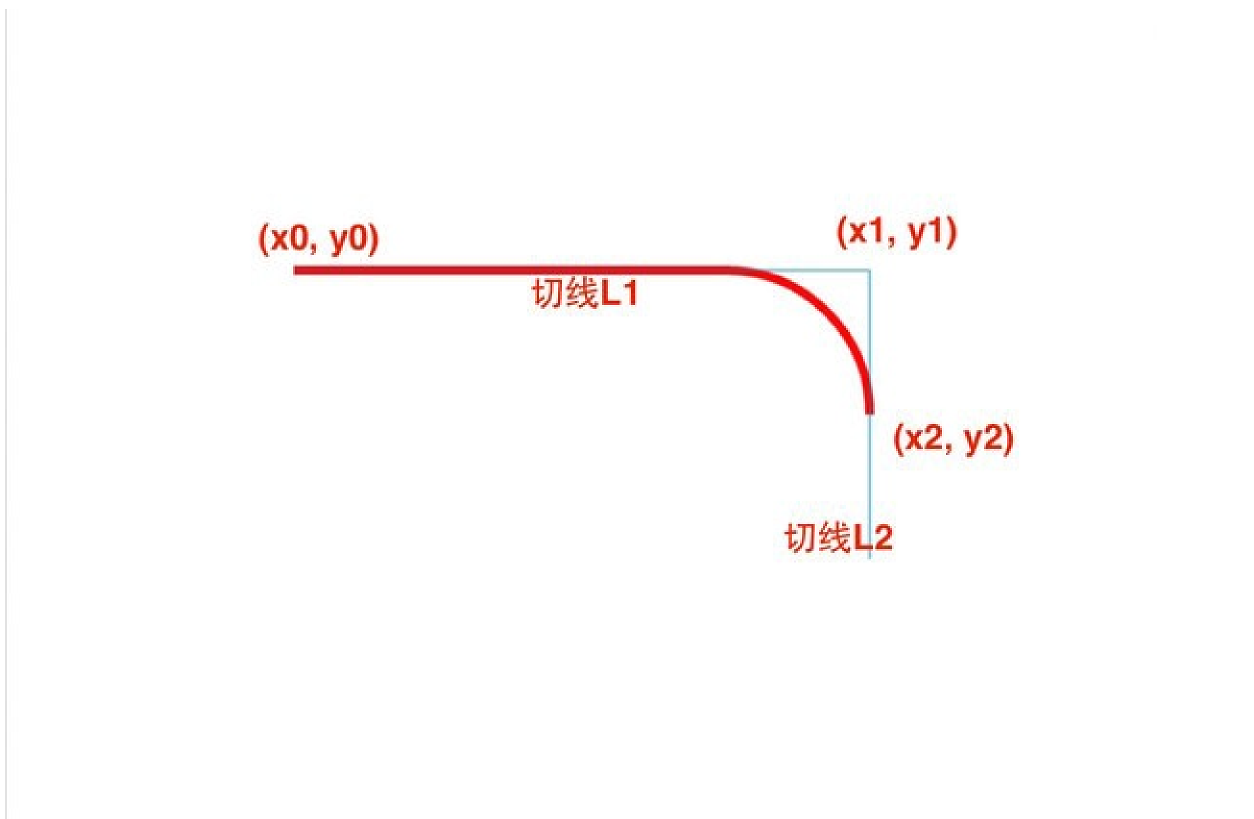
    }
</script>
</body>
</html>
```

演示 10-1

运行结果：



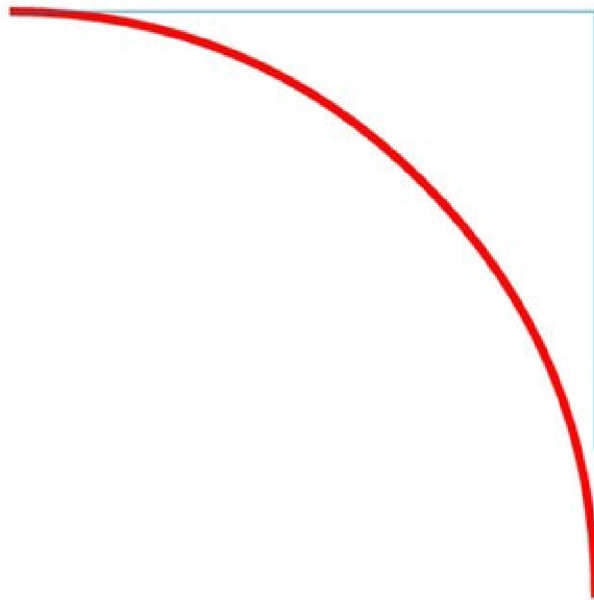
这个案例也很好说明了 `arcTo()` 的各个关键点的作用。为了更清楚的解释，我再标注一个分析图。



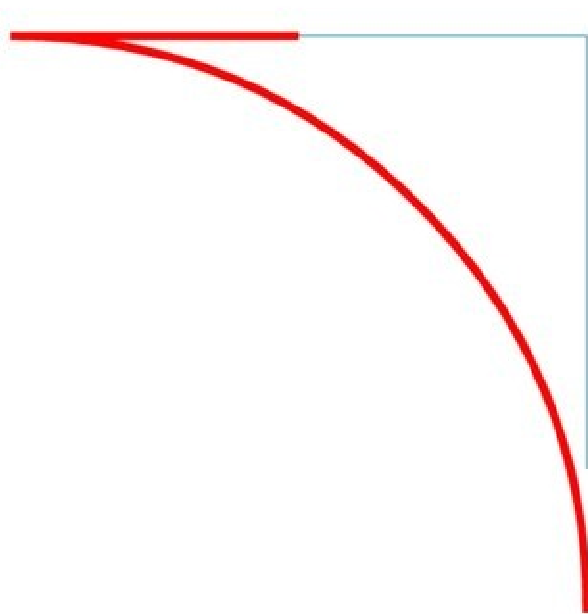
这里注意一下， `arcTo()` 绘制的起点是(x0, y0)，但(x0, y0)不一定是圆弧的切点。真正的 `arcTo()` 函数只传入(x1, y1)和(x2, y2)。其中(x1, y1)称为控制点，(x2, y2)是圆弧终点的切点，它不一定在圆弧上。但(x0, y0)一定在圆弧上。

有一点点绕，下面我们改变 `drawArcTo()` 函数的参数来试验一下。

- (x2, y2)不一定在弧线上： `drawArcTo(context, 200, 100, 600, 100, 600, 400, 400);`



- (x0, y0)一定在弧线上： `drawArcTo(context, 400, 100, 600, 100, 600, 400, 400);`



挺有意思的，它为了经过(x0, y0)直接将切点和(x0, y0)连接起来形成线段。好执着的弧线……

绘制微信对话框

大家可以尝试着使用Canvas绘制一下微信聊天界面，作为练习与巩固。



这里使用到了绘制矩形，绘制圆角矩形，绘制多线条图形，填充颜色的一些知识。还有一些 Canvas文本API 我们并没有说到，所以大家只要能绘制出一个大概的界面就算合格了。能够绘制出来，也就基本掌握了Canvas API。

其实上述对话是生成出来的——“微信界面生成器网页版”，可谓是微商神器。是不是非常的酷？



这只是暑假花两天时间写的最初版本，还尚未达到发布的地步，在我写本节的时候，这个网页的界面还正在优化中。大家可以尝试自己动手做做，也可以关注和参考我的这个小项目github：微信界面生成器。本节就不再重复给出界面代码了。

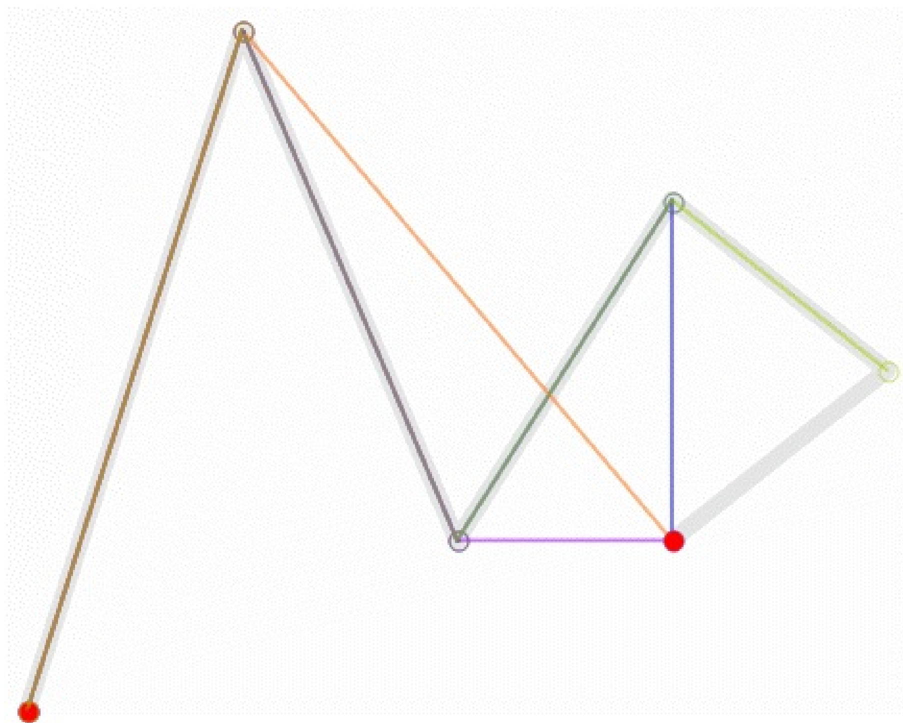
好了，学到这里基本上已经学完了所有基本的Canvas绘图的api，大家拿起自己的画笔，自由的发挥吧！😋

Ch11 二次贝塞尔曲线

贝塞尔曲线

Bézier curve(贝塞尔曲线)是应用于二维图形应用程序的数学曲线。 曲线定义：起始点、终止点、控制点。通过调整控制点，贝塞尔曲线的形状会发生变化。 1962年，法国数学家Pierre Bézier第一个研究了这种矢量绘制曲线的方法，并给出了详细的计算公式，因此按照这样的公式绘制出来的曲线就用他的姓氏来命名，称为贝塞尔曲线。

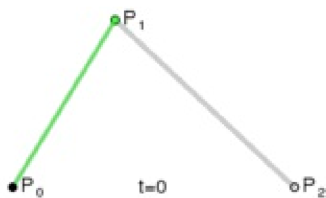
这里我们不介绍计算公式，只要知道贝塞尔曲线是一条由起始点、终止点和控制点所确定的曲线就行了。而**n**阶贝塞尔曲线就有**n-1**个控制点。用过Photoshop等绘图软件的同学应该比较熟悉，因为其中的钢笔工具设置锚点绘制路径的时候，用到的就是贝塞尔曲线。下图就是五阶贝塞尔曲线的绘制过程。



是不是非常的酷炫？

二次贝塞尔曲线

都介绍了五次贝塞尔曲线，那二次的肯定不在话下了。大家一定能想象出它长啥样。没错，就是下面这样。



在Canvas里，二次贝塞尔曲线的方法如下。

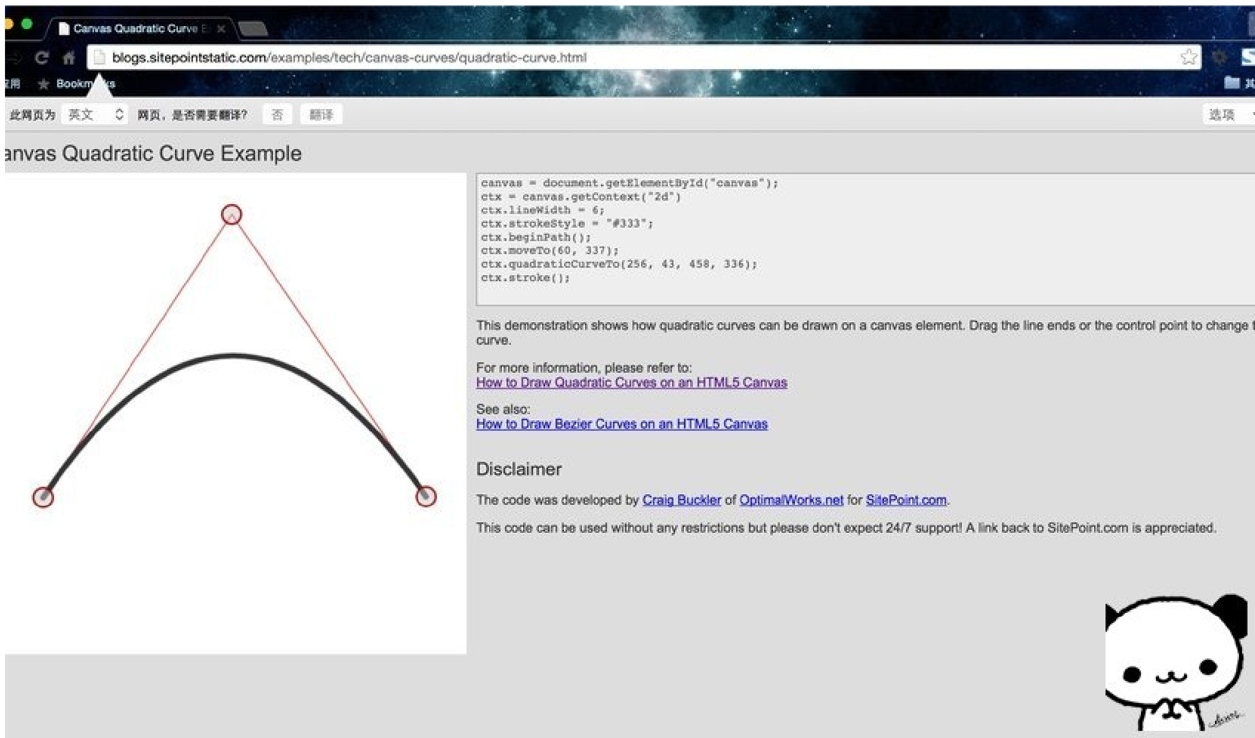
```
context.quadraticCurveTo(cpx,cpy,x,y);
```

这里和 `acrTo()` 有异曲同工之妙。P0是起始点，所以通常搭配 `moveTo()` 或 `lineTo()` 使用。P1(cpx, cpy)是控制点，P2(x, y)是终止点，它们不是相切的关系。什么关系呢？如果偏要问，我只好给出下面的公式……

$$B(t)=\sum_{i=0}^{n}\binom{n}{i}Pi(1-t)^{n-i}t^{i}=\binom{n}{0}P0(1-t)^{n}t^{0}+\binom{n}{1}P1(1-t)^{n-1}t^{1}+\ldots+\binom{n}{n-1}Pn-1(1-t)^{1}t^{n-1}+\binom{n}{n}Pn(1-t)^{0}t^{n},t\in[0,1]$$

这么复杂的函数式，那我们绘图时， `quadraticCurveTo(cpx,cpy,x,y)` 的参数怎么填？很简单，可以简单调试直至得到你想要的效果。或者使用一些工具。

这里我提供一个很不错的在线转换器，界面如下。



这里我把三个控制点调好，变成一个大山的形状，右侧自动生成了代码，我们只要复制就行了。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>二次贝塞尔曲线</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

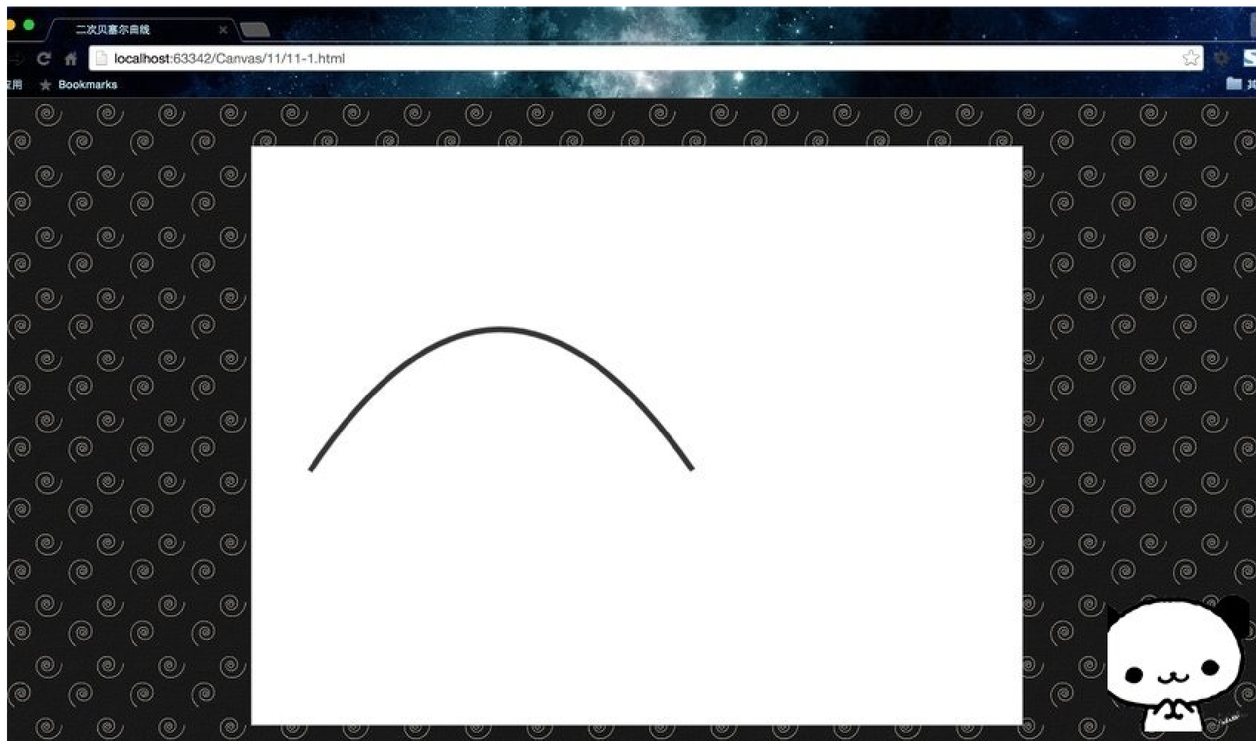
<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.lineWidth = 6;
        context.strokeStyle = "#333";
        context.beginPath();
        context.moveTo(60, 337);
        context.quadraticCurveTo(256, 43, 458, 336);
        context.stroke();
    };

</script>
</body>
</html>
```

演示 11-1

运行结果：



这样我们把在线转换工具里的贝塞尔曲线搬进我们自己的画布里了，是不是非常的酷？大家如果有特别难的曲线没法用 `arcTo()` 绘制，就可以尝试一下使用这个工具绘制贝塞尔曲线。

本节的内容非常少，童鞋们不要停下脚步，整理好行装，一并把最终BOSS——三次贝塞尔曲线消灭掉！打败他之后，我们就是初级艺术家了，是不是非常的兴奋？让我们继续前进！😋

Ch12 三次贝塞尔曲线

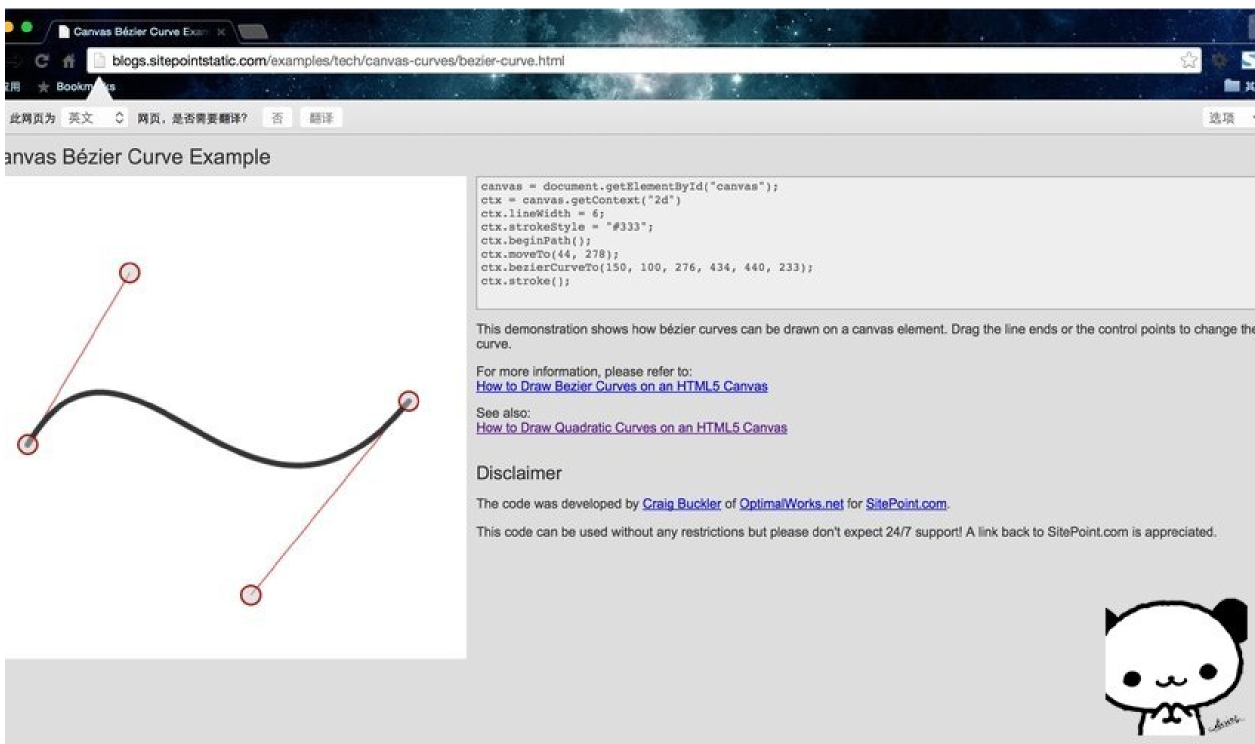
`bezierCurveTo()` 方法

绘制三次贝塞尔曲线代码如下。

```
context.bezierCurveTo(cp1x,cp1y,cp2x,cp2y,x,y);
```

这个方法可谓是绘制波浪线的神器。根据之前的结论，n阶贝塞尔曲线就有n-1个控制点，所以三次贝塞尔曲线有1个起始点、1个终止点、2个控制点。因此传入的6个参数分别为控制点cp1 (cp1x, cp1y)，控制点cp2 (cp2x, cp2y)，与终止点 (x, y)。

这个方法也是不用大家去掌握参数具体是怎么填的，只要知道参数的意义就行。和 `quadraticCurveTo()` 方法一样， `bezierCurveTo()` 的三次贝塞尔曲线网上也能找到互动的网页工具。这里提供一个网页：Canvas Bézier Curve Example，大家可以动手试一下。



绘制XP壁纸

这里我们拿XP的壁纸开刀，来练习一下我们之前学习过的绘制方法。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>XP壁纸</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        drawPrairie(context);
        drawSky(context);
        for(var i=0; i <5; i++){
            var x0 = 500 * Math.random() + 50;
            var y0 = 200 * Math.random() + 50;
            var c0 = 100 * Math.random() + 50;
            drawCloud(context, x0, y0, c0);

        }

    };
```

```
    function drawSky(cxt){
        cxt.save();

        cxt.beginPath();
        cxt.moveTo(0, 420);
        cxt.bezierCurveTo(250, 300, 350, 550, 800, 400);
        cxt.lineTo(800,0);
        cxt.lineTo(0,0);
        cxt.closePath();

        var lineStyle = cxt.createRadialGradient(400, 0, 50, 400
, 0, 200);
        lineStyle .addColorStop(0, "#42A9AA");
        lineStyle .addColorStop(1, "#2491AA");

        cxt.fillStyle = lineStyle;

        cxt.fill();

        cxt.restore();
    }

    function drawPrairie(cxt){
        cxt.save();

        cxt.beginPath();
        cxt.moveTo(0, 420);
        cxt.bezierCurveTo(250, 300, 350, 550, 800, 400);
        cxt.lineTo(800,600);
        cxt.lineTo(0,600);
        cxt.closePath();

        var lineStyle = cxt.createLinearGradient(0, 600, 600, 0)
;
        lineStyle .addColorStop(0, "#00AA58");
        lineStyle .addColorStop(0.3, "#63AA7B");
        lineStyle .addColorStop(1, "#04AA00");

        cxt.fillStyle = lineStyle;
```

```
        cxt.fill();

        cxt.restore();
    }




    /*渲染单个云朵
     context:  canvas.getContext("2d")对象
     cx: 云朵X轴位置
     cy: 云朵Y轴位置
     cw: 云朵宽度
     */
    function drawCloud(cxt, cx, cy, cw) {
        //云朵移动范围即画布宽度
        var maxWidth = 800;
        //如果超过边界从头开始绘制
        cx = cx % maxWidth;
        //云朵高度为宽度的60%
        var ch = cw * 0.6;
        //开始绘制云朵

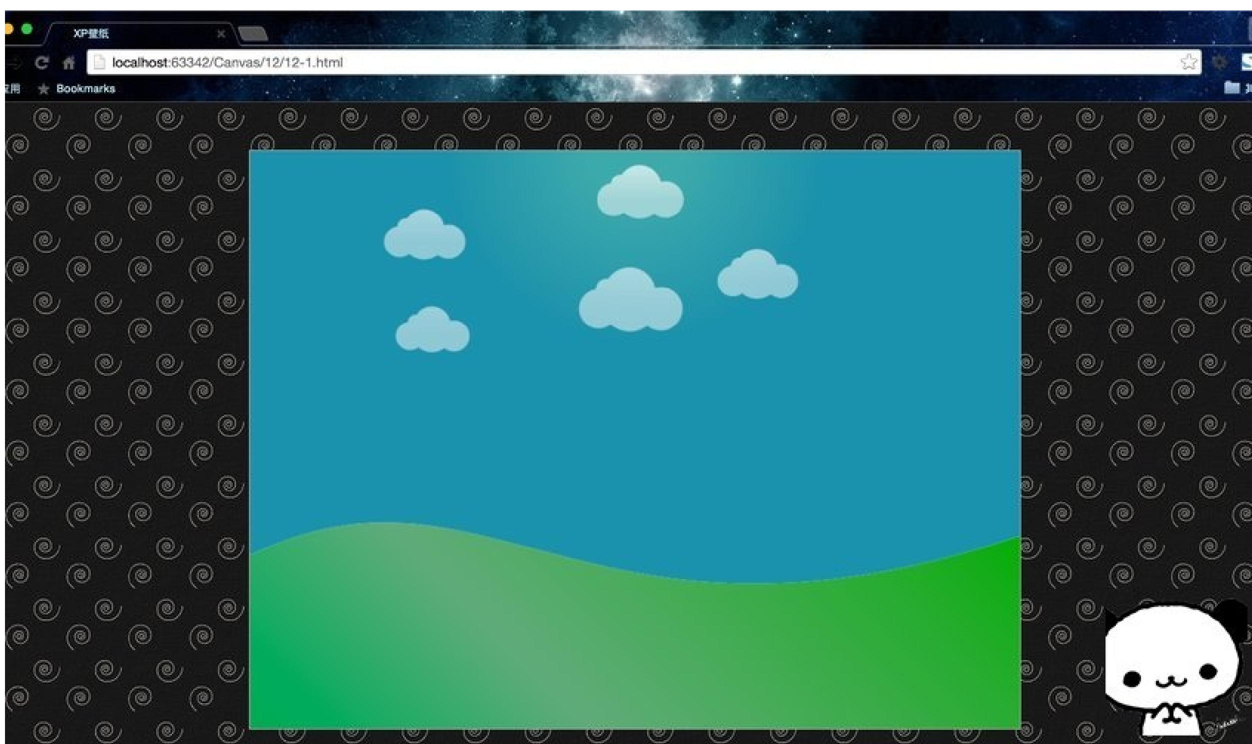
        cxt.beginPath();
        cxt.fillStyle = "white";
        //创建渐变
        var grd = cxt.createLinearGradient(0, 0, 0, cy);
        grd.addColorStop(0, 'rgba(255,255,255,0.8)');
        grd.addColorStop(1, 'rgba(255,255,255,0.5)');
        cxt.fillStyle = grd;

        //在不同位置创建5个圆拼接成云朵现状
        cxt.arc(cx, cy, cw * 0.19, 0, 360, false);
        cxt.arc(cx + cw * 0.08, cy - ch * 0.3, cw * 0.11, 0, 360
, false);
        cxt.arc(cx + cw * 0.3, cy - ch * 0.25, cw * 0.25, 0, 360
, false);
        cxt.arc(cx + cw * 0.6, cy, cw * 0.21, 0, 360, false);
        cxt.arc(cx + cw * 0.3, cy - ch * 0.1, cw * 0.28, 0, 360,
false);
        cxt.closePath();
```

```
        cxt.fill();
    }
</script>
</body>
</html>
```

演示 12-1

运行结果：



是不是很萌？是不是非常的酷！这个案例几乎用到了之前所传授给你们的所有武功——三次贝塞尔曲线，径向渐变，线性渐变，绘制圆弧等等。分开写了三个函数，一个绘制草原、一个绘制蓝天、一个绘制白云......大家尝试自己实现一下，当做一次阶段性复习~

保存和恢复Canvas状态

这里还使用到了两个新方法 `save()` 和 `restore()` 。之前说过了canvas是基于状态的绘制（说了好多次，感觉自己好啰嗦）。保存（推送）当前状态到堆栈，调用以下函数。

```
context.save();
```

调出最后存储的堆栈恢复画布，使用以下函数。

```
context.restore();
```

不知道大家壁纸绘制的如何，肯定非常的酷有没有？到此为止路径的知识和填充样式我们已经全部讲完了，大家也画出了很多或优美、或抽象的艺术作品。不管怎么样，这是属于我们的艺术，我们继续前进！😋

Ch13 平移变换

图形变换

从今天开始，我们就开始谈一谈图形变换。图形变换是指用数学方法调整所绘形状的物理属性，其实质是坐标变形。所有的变换都依赖于后台的数学矩阵运算，所以我们只要使用变换的功能即可，无需去理解这些运算。谈到图形变换，不得不得说的三个基本变换方法就是：

1. 平移变换：`translate(x,y)`
2. 旋转变换：`rotate(deg)`
3. 缩放变换：`scale(sx,sy)`

其实坐标变形的本质是变换矩阵，所以在最后我们会谈一谈一个万能的变换方法——矩阵变换 `transform()` 。那么，我们按部就班，这一节我们来谈一谈平移变换。

平移变换 `translate()`

平移变换，故名思议，就是一般的图形位移。比如这里我想将位于（100，100）的矩形平移至（200，200）点。那么我只要在绘制矩形之前加上 `context.translate(100,100)` 即可。

这里的 `translate()` 只传入两个参数，其实就是新画布坐标系原点的坐标。下面结合代码来看看效果。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>平移变换</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

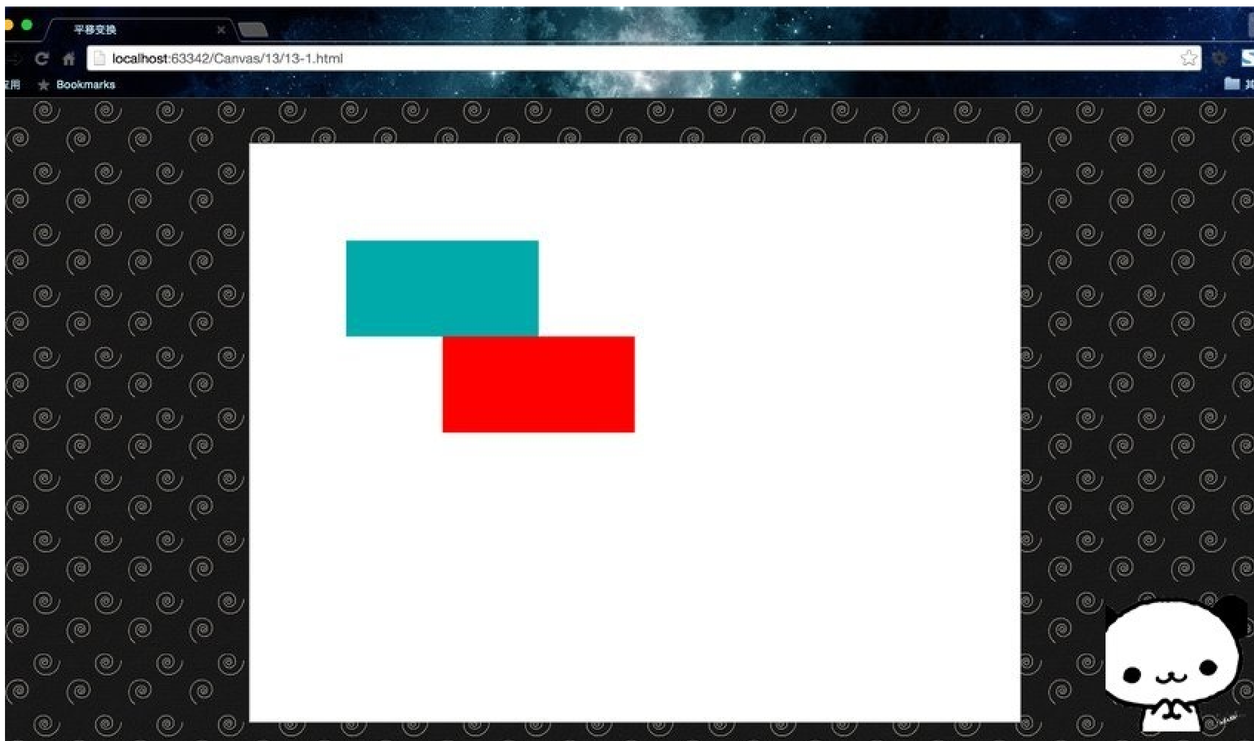
        context.fillStyle = "#00AAAA";
        context.fillRect(100,100,200,100);

        context.fillStyle = "red";
        context.translate(100,100);
        context.fillRect(100,100,200,100);

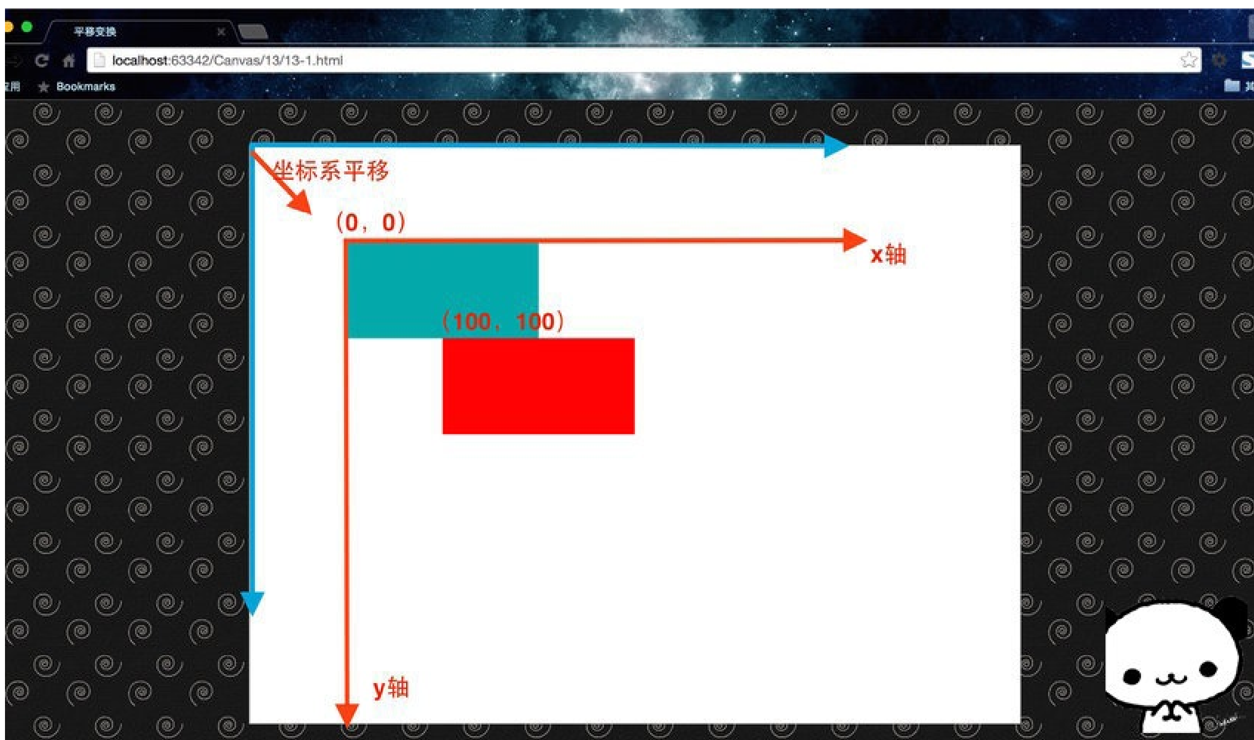
    };
</script>
</body>
</html>
```

演示 13-1

运行结果：



这里的蓝色矩形，是矩形原来的位置，然后调用 `translate()` 方法，将矩形位移至（200，200），即红色矩形的位置。我们来用一张图看看，它是怎么做到平移变换的。



没错，其实这里的平移变换实质就是在平移坐标系，而对 `translate()` 传入的参数，实质就是新坐标系相对于旧坐标系的原点。这使得我们依旧是在（100，100）绘制的红色矩形，在平移坐标系之后，变到了（200，200）处。

注意使用状态保存

其实这里有一个坑，我们如果想把矩形平移至（300，300）怎么办呢？或许我们会想，直接调用 `context.translate(200,200)` 就可以了。好，我们看看效果。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>平移变换</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.fillStyle = "#00AAAA";
        context.fillRect(100,100,200,100);
```

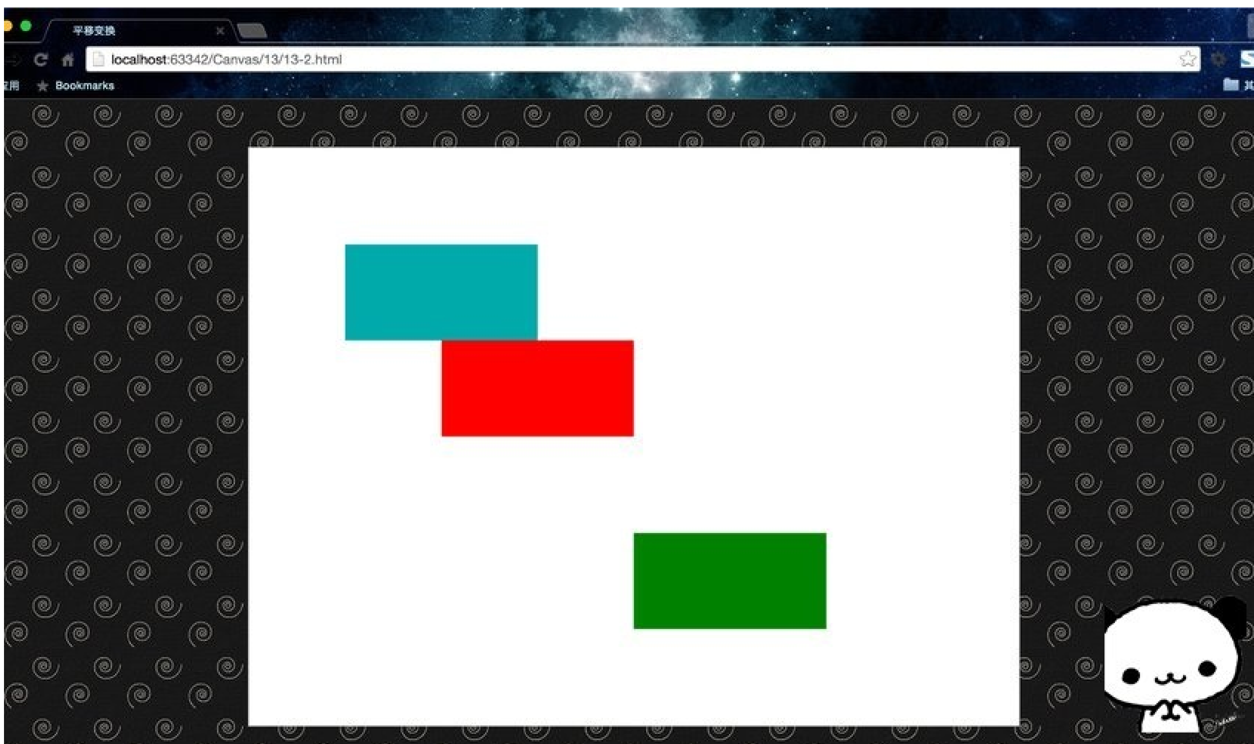
```
            context.fillStyle = "red";
            context.translate(100,100);
            context.fillRect(100,100,200,100);

            context.fillStyle = "green";
            context.translate(200,200);
            context.fillRect(100,100,200,100);

        };
</script>
</body>
</html>
```

演示 13-2

运行结果：



这里的绿色矩形并没有如我们所愿在（300，300）位置处，而是跑到了（400，400）这里。为什么呢？想必大家已经知道了答案——Canvas是基于状态的绘制。在我们第一次平移之后，坐标系已经在（100，100）处了，所以如果继续平移，这个再基于新坐标系继续平移坐标系。那么要怎么去解决呢？很简单，有两个方法。

第一，在每次使用完变换之后，记得将坐标系平移回原点，即调用 `translate(-x,-y)` 。

第二，在每次平移之前使用 `context.save()` ，在每次绘制之后，使用 `context.restore()` 。

切记，千万不要再想着我继续紧接着第一次平移之后再平移 `translate(100,100)` 不就行了，这样你自己的坐标系统就会乱套，根本找不到自己的坐标系原点在哪，在多次变换或者封装函数之后，会坑死你。所以一定要以最初状态为最根本的参照物，这是原则性问题。这里我建议使用第二种方法，而且在涉及所有图形变换的时候，都要这么处理，不仅仅是平移变换。

具体使用如下。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>平移变换</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; margin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.fillStyle = "#00AAAA";
        context.fillRect(100,100,200,100);
```

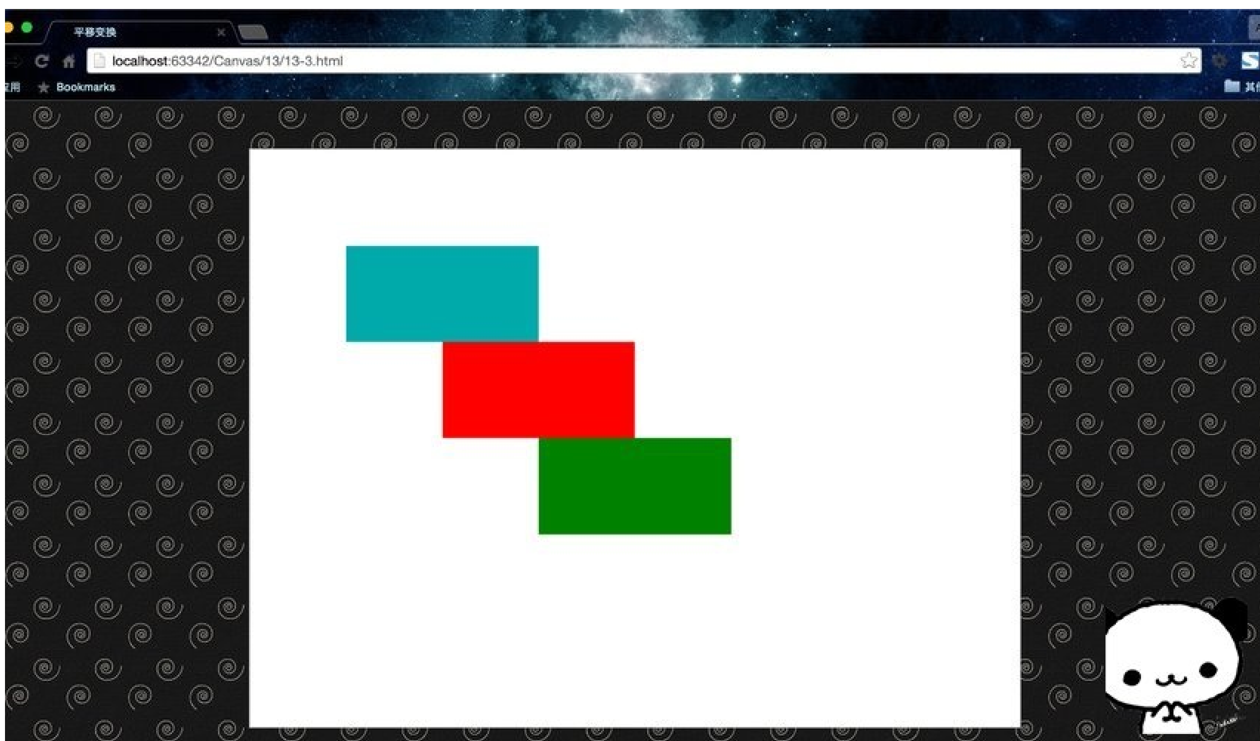
```
        context.save();
        context.fillStyle = "red";
        context.translate(100,100);
        context.fillRect(100,100,200,100);
        context.restore();

        context.save();
        context.fillStyle = "green";
        context.translate(200,200);
        context.fillRect(100,100,200,100);
        context.restore();

    };
</script>
</body>
</html>
```

演示 13-3

运行结果：



因此，在使用图形变换的时候，要记得结合使用状态保存。

好了，这一节的内容很少，我们继续前进。😋

Ch14 旋转变换

旋转变换 `rotate()`

同画圆弧一样，这里的 `rotate(deg)` 传入的参数是弧度，不是角度。同时需要注意的是，这个的旋转是以坐标系的原点（0，0）为圆心进行的顺时针旋转。所以，在使用 `rotate()` 之前，通常需要配合使用 `translate()` 平移坐标系，确定旋转的圆心。即，旋转变换通常搭配平移变换使用的。

最后一点需要注意的是，Canvas是基于状态的绘制，所以每次旋转都是接着上次旋转的基础上继续旋转，所以在使用图形变换的时候必须搭配 `save()` 与 `restore()` 方法，一方面重置旋转角度，另一方面重置坐标系原点。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>旋转变换</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
```

```
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

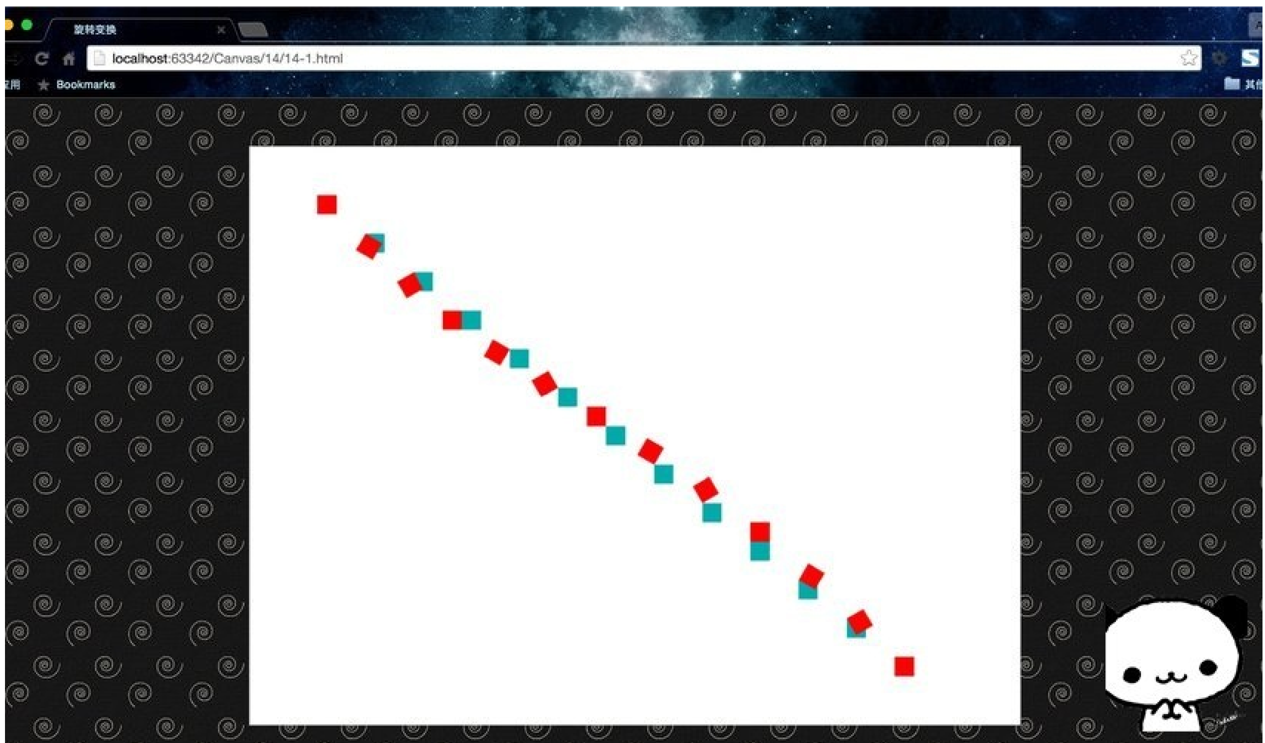

        for(var i = 0; i <= 12; i++){
            context.save();
            context.translate(70 + i * 50, 50 + i * 40);
            context.fillStyle = "#00AAAA";
            context.fillRect(0,0,20,20);
            context.restore();

            context.save();
            context.translate(70 + i * 50, 50 + i * 40);
            context.rotate(i * 30 * Math.PI / 180);
            context.fillStyle = "red";
            context.fillRect(0,0,20,20);
            context.restore();
        }

    };
</script>
</body>
</html>
```

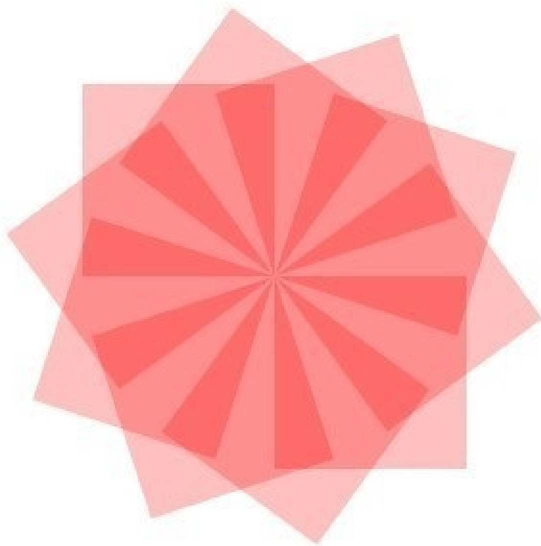
演示 14-1

运行结果：



这里用for循环绘制了14对正方形，其中蓝色是旋转前的正方形，红色是旋转后的正方形。每次旋转都以正方形左上角顶点为原点进行旋转。每次绘制都被 `save()` 与 `restore()` 包裹起来，每次旋转前都移动了坐标系。童鞋们可以自己动动手，实践一下，就能体会到旋转变换的奥妙了。

绘制魔性Logo



这是在度娘上看到了一个logo，巧妙运用了旋转变换，用一个很简单矩形，通过旋转变换，变成了一个很漂亮的logo。这logo是不是很有魔性？童鞋们动动脑，尝试实现一下它。下面，提供我实现它的代码。

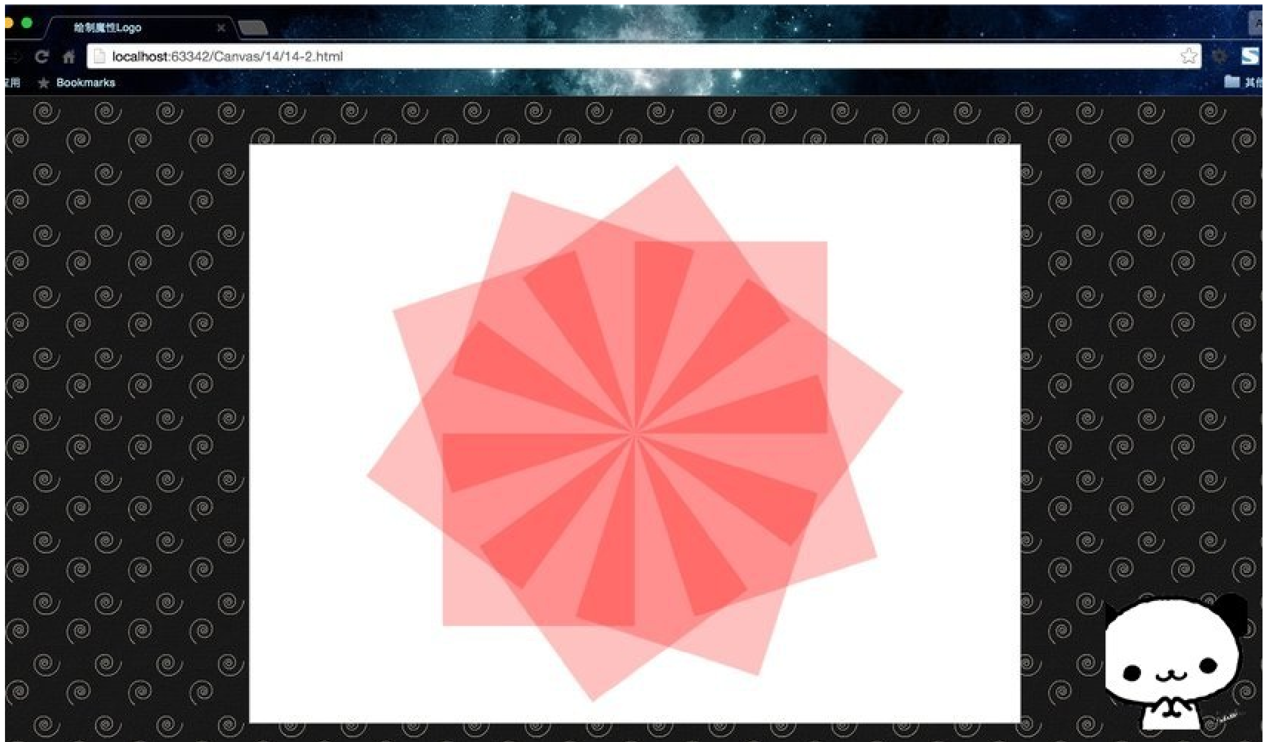
```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制魔性Logo</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        for(var i=1; i<=10; i++){
            context.save();
            context.translate(400,300);
            context.rotate(36 * i * Math.PI / 180);
            context.fillStyle = "rgba(255,0,0,0.25)";
            context.fillRect(0, -200, 200, 200);
            context.restore();
        }
    };
</script>
</body>
</html>
```

演示 14-2

运行结果：



是不是非常的酷？这个图形稍微分析一下发现还是蛮简单的，就是让一个正放形，以屏幕中点（即初始正方形左下角顶点）为圆心进行旋转。

艺术是不是很美妙？大家一定以及体会到了Canvas的奇妙，简简单单的几行代码就能实现无穷无尽的效果。只要脑洞够大，没有什么是不可以实现的。所以，扬起咱们的艺术家的旗帜，加快步伐，继续前进！😋

Ch15 缩放变换

缩放变换 `scale()`

缩放变换 `scale(sx,sy)` 传入两个参数，分别是水平方向和垂直方向上对象的缩放倍数。例如 `context.scale(2,2)` 就是对图像放大两倍。其实，看上去简单，实际用起来还是有一些问题的。我们来看一段代码。

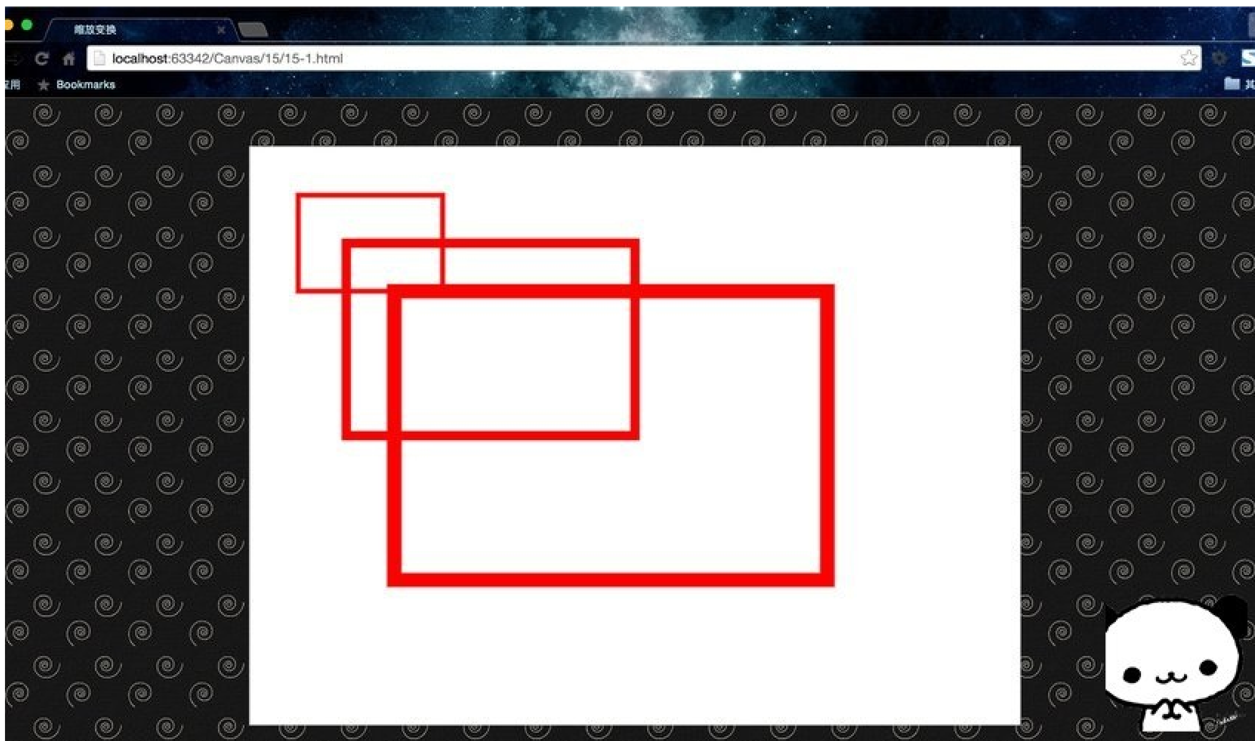
```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>缩放变换</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.strokeStyle = "red";
        context.lineWidth = 5;
        for(var i = 1; i < 4; i++){
            context.save();
            context.scale(i,i);
            context.strokeRect(50,50,150,100);
            context.restore();
        }
    };
</script>
</body>
</html>
```

演示 15-1

运行结果：



缩放变换应注意的问题

看了上面的例子，大家一定对产生的结果有点奇怪。一是左上角顶点的坐标变了，而是线条的粗细也变了。因此，对于缩放变换有两点问题需要注意：

1. 缩放时，图像左上角坐标的位置也会对应缩放。
2. 缩放时，图像线条的粗细也会对应缩放。

比如对于最小的那个原始矩形，它左上角的坐标是（50，50），线条宽度是5px，但是放大2倍后，左上角坐标变成了（100，100），线条宽度变成了10px。这就是缩放变换的副作用。

童鞋们一定在期待着我说解决副作用的途径。很遗憾，没有什么好的方法去解决这些副作用。如果想固定左上角坐标缩放，可以把左上角坐标变成（0，0），这样的话无论是什么倍数，0乘上它还是0，所以不变。如果不想让线条粗细变化，那就别使用线条。或者自己封装一个函数，不要使用 `scale()` 。

究其根本，之前我们说过平移变换、旋转变换、缩放变换都属于坐标变换，或者说是画布变换。因此，缩放并非缩放的是图像，而是整个坐标系、整个画布！就像是对坐标系的单位距离缩放了一样，所以坐标和线条都会进行缩放。仔细想想，这一

切貌似挺神奇的。

好啦，这一节的内容就这么少，让我们检查一下弹药是否充足，准备征战图形变换的最终Boss。继续前进！😋

Ch16 矩阵变换

变换矩阵

之前三节所说的坐标变换的三种方式——平移 `translate()` ，缩放 `scale()` ，以及旋转 `rotate()` 都可以通过 `transform()` 做到。

在介绍矩阵变换 `transform()` 前，我们来说一说什么是变换矩阵。

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

以上是Canvas中 `transform()` 方法所对应的变换矩阵。而此方法正是传入图中所示的六个参数，具体为 `context.transform(a,b,c,d,e,f)` 。

各参数意义对应如下表：

参数	意义
a	水平缩放(1)
b	水平倾斜(0)
c	垂直倾斜(0)
d	垂直缩放(1)
e	水平位移(0)
f	垂直位移(0)

当我们把对应的0或1代入进矩阵，可以发现这是一个单位矩阵（水平和垂直缩放默认值是1，代表缩放1倍，即不缩放）。该方法使用一个新的变化矩阵与当前变换矩阵进行乘法运算，然后得到各种变化的效果。

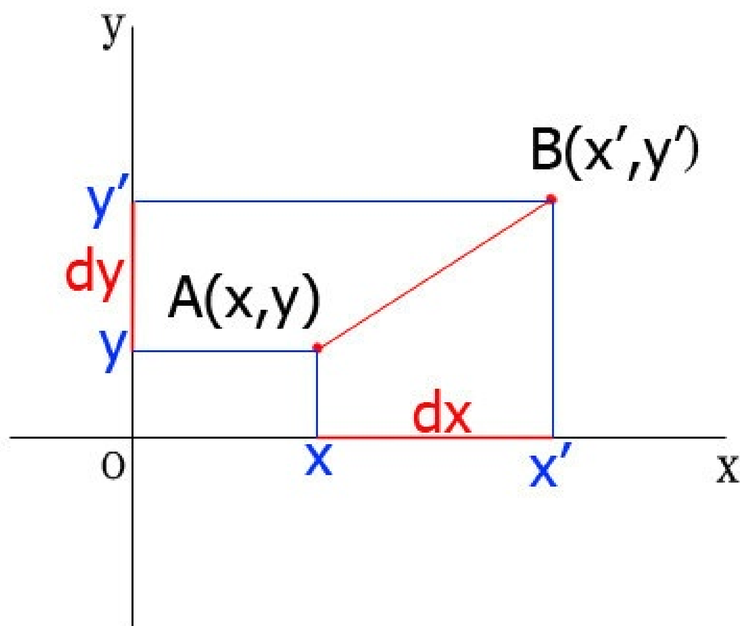
这里简单说一下，当我们想对一个图形进行变换的时候，只要对变换矩阵相应的参数进行操作，操作之后，对图形的各个定点的坐标分别乘以这个矩阵，就能得到新的定点的坐标。

而Canvas绘图中，就给咱们提供了一个函数来改变这个变换矩阵，那就是 `transform()` 。

矩阵变换 `transform()`

> 注：以下三个变换的转换结论转自张雯的博客，童鞋们可以戳进去看看详情。

平移变换



如上图所示： `x’ = x + dx` ， `y’ = y + dy` 。

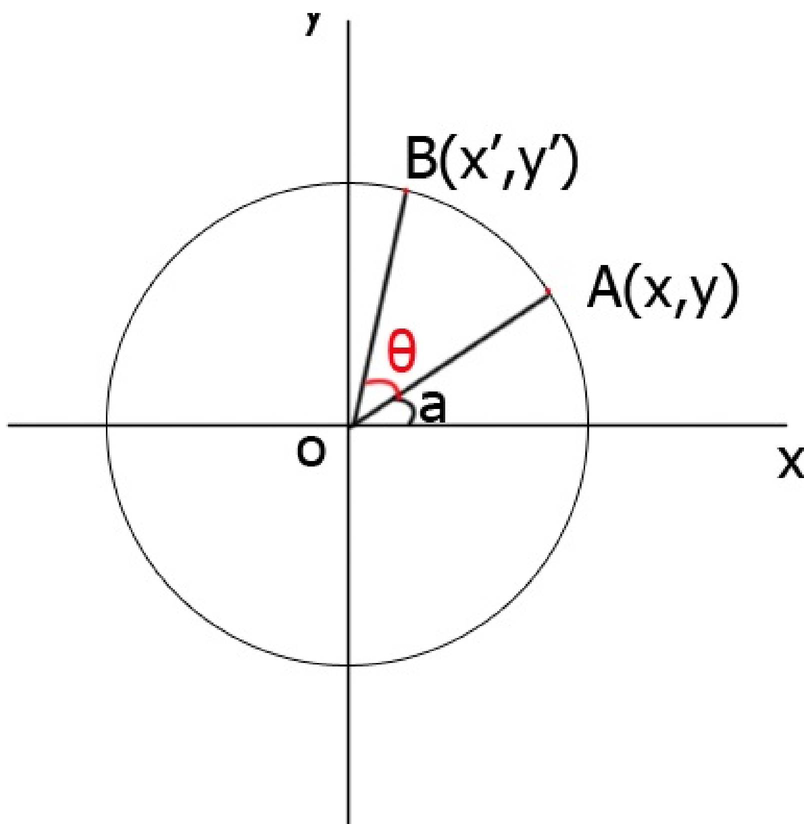
也即是说可以使用 `context.transform (1,0,0,1,dx,dy)` 代替 `context.translate(dx,dy)` 。也可以使用 `context.transform(0,1,1,0,dx,dy)` 代替。

缩放变换

同理可以使用 `context.transform(sx,0,0,sy,0,0)` 代替 `context.scale(sx, sy)`；

也可以使用 `context.transform(0,sy,sx,0,0,0)`；

旋转变换



如上图图所示：

B点是通过A点逆时针旋转θ得到，`x = r * cosa`，`y = r * sina`

即 `x′=r*cos(a+θ)=x*cosθ-y*sinθ`，`y′=r*sin(a+θ)=x*sinθ+y*cosθ`

也即是 $$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

综上：`context.transform(Math.cos(θ*Math.PI/180)，Math.sin(θ*Math.PI/180), -Math.sin(θ*Math.PI/180),Math.cos(θ*Math.PI/180)，0，0)` 可以替代 `context.rotate(θ)`。

也可以使用 `context.transform(-Math.sin(θ*Math.PI/180),Math.cos(θ*Math.PI/180)，Math.cos(θ*Math.PI/180)，Math.sin(θ*Math.PI/180)，0，0)` 替代。

以上结论引自张雯的博客，很好的说明了矩阵变换的过程，但是结论过于复杂，建议使用 `transform()` 的时候，可以在如下几个情况下使用：

1. 使用 `context.transform (1,0,0,1,dx,dy)` 代替 `context.translate(dx,dy)`
2. 使用 `context.transform(sx,0,0,sy,0,0)` 代替 `context.scale(sx, sy)`
3. 使用 `context.transform(0,b,c,0,0,0)` 来实现倾斜效果(最实用)。

不用再使用它去实现旋转了，另外也没有也不用全记这些结论，直接记下abcdef六个参数的意义，效果是一样的。

下面我们看一个代码，熟悉一下：

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>矩阵变换</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
```

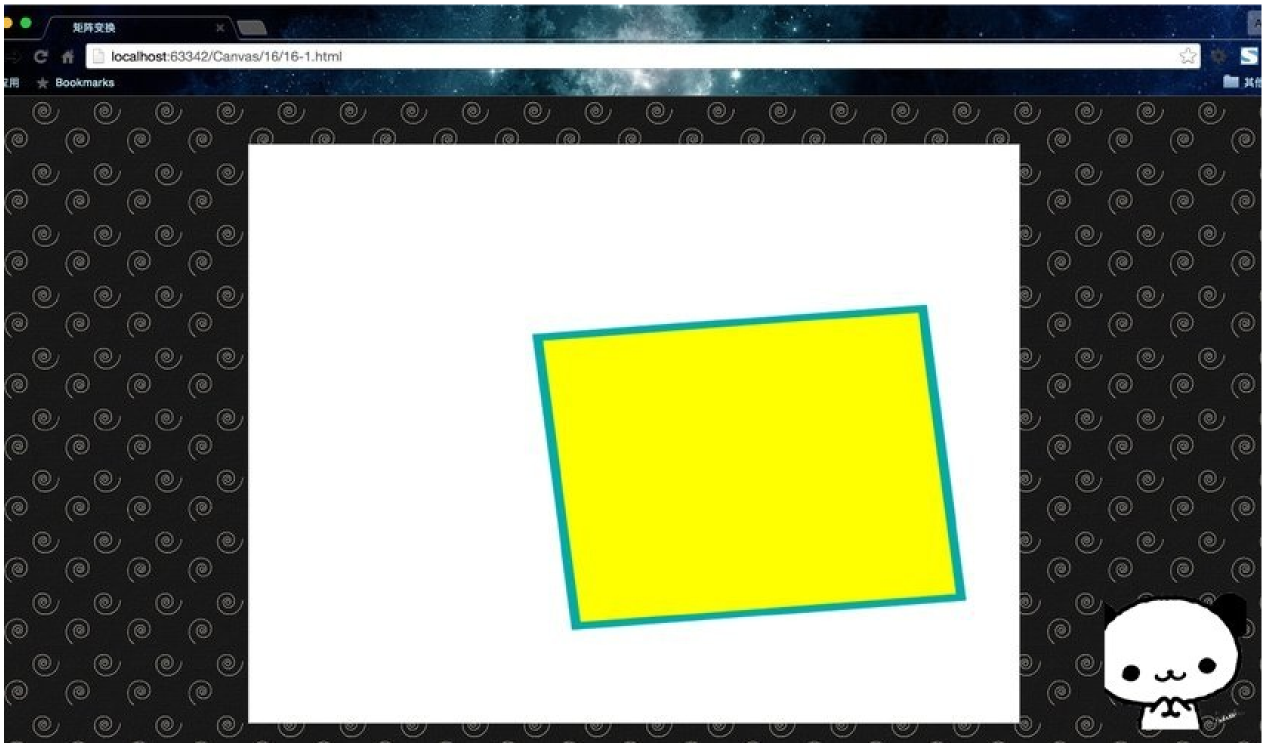
```
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.fillStyle = "yellow";
        context.strokeStyle = "#00AAAA";
        context.lineWidth = 5;

        context.save();
        //平移至(300,200)
        context.transform(1,0,0,1,300,200);
        //水平方向放大2倍，垂直方向放大1.5倍
        context.transform(2,0,0,1.5,0,0);
        //水平方向向右倾斜宽度10%的距离，垂直方向向上倾斜高度10%的距离
        context.transform(1,-0.1,0.1,1,0,0);
        context.fillRect(0,0,200,200);
        context.strokeRect(0,0,200,200);
        context.restore();
    };
</script>
</body>
</html>
```

演示 16-1

运行结果：



`setTransform()` 方法

`transform()` 方法的行为相对于由 `rotate()` , `scale()` , `translate()` , or `transform()` 完成的其他变换。例如：如果我们已经将绘图设置为放到两倍，则 `transform()` 方法会把绘图放大两倍，那么我们的绘图最终将放大四倍。这一点和之前的变换是一样的。

但是 `setTransform()` 不会相对于其他变换来发生行为。它的参数也是六个， `context.setTransform(a,b,c,d,e,f)` ，与 `transform()` 一样。

这里我们通过一个例子来说明： 绘制一个矩形，通过 `setTransform()` 重置并创建新的变换矩阵，再次绘制矩形，重置并创建新的变换矩阵，然后再次绘制矩形。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>矩阵变换</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
```

```
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

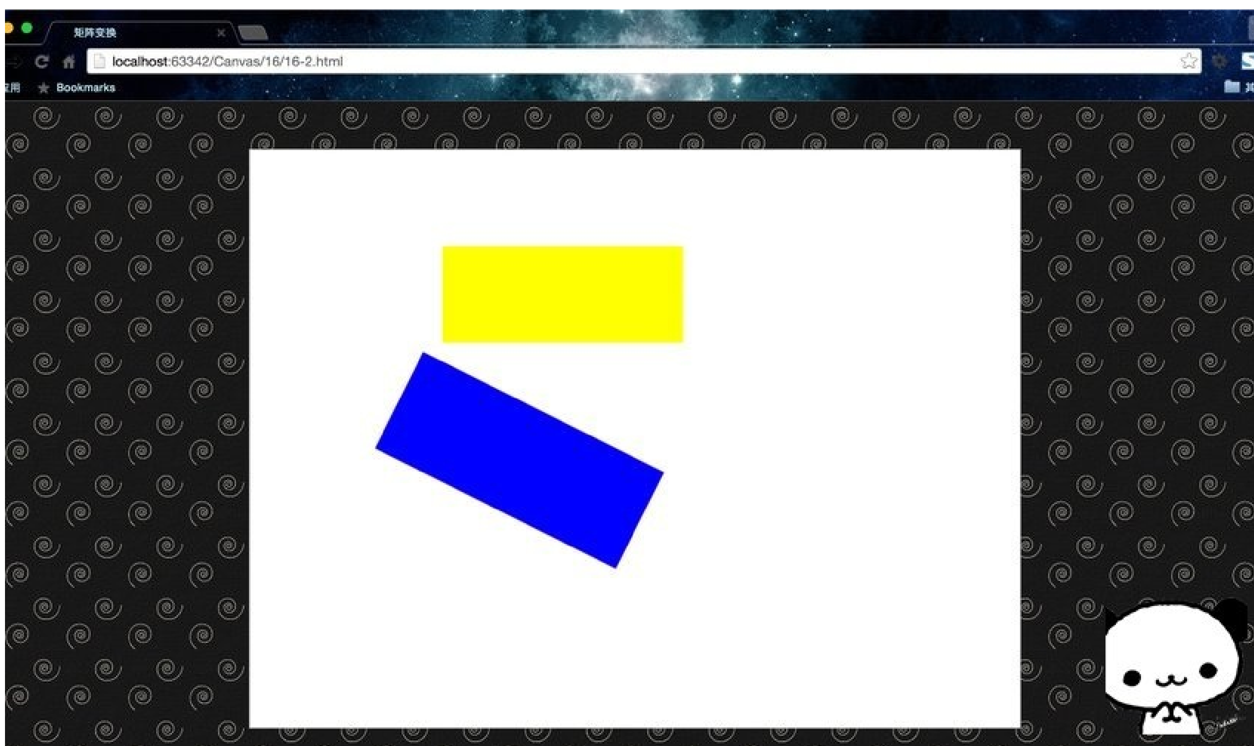
        context.fillStyle="yellow";
        context.fillRect(200,100,250,100)

        context.setTransform(1,0.5,-0.5,1,30,10);
        context.fillStyle="red";
        context.fillRect(200,100,250,100);

        context.setTransform(1,0.5,-0.5,1,30,10);
        context.fillStyle="blue";
        context.fillRect(200,100,250,100);
    };
</script>
</body>
</html>
```

演示 16-2

运行结果：



解释一下过程：每当我们调用 `setTransform()` 时，它都会重置前一个变换矩阵然后构建新的矩阵，因此在下面的例子中，不会显示红色矩形，因为它在蓝色矩形下面。

这一节的内容有些难和多，但是除了倾斜其他并不实用，所以童鞋们尽量了解一下就好，能够掌握就更好啦！至此，所有的图形变换我们已经上完了，基本的三大变化——平移变换、旋转变换、缩放变换，以及万能的矩阵变换。大家都以及完全掌握了吗？如果不熟练，要回头多看看，并且多加练习。稍作休息，下一节开始，就是新的航程！我们继续前进！😋

Ch17 文本显示与渲染

文本 API 简介

今天我们开始征战一个全新的内容——HTML5 Canvas的文本API！要知道，艺术家通常同时也是一个书法家，所以我们要学习写字，而且是写出漂亮的字。是不是很有意思？

好了，先预告一下Canvas 文本API有哪些。

属性	描述
`font`	设置或返回文本内容的当前字体属性
`textAlign`	设置或返回文本内容的当前对齐方式
`textBaseline`	设置或返回在绘制文本时使用的当前文本基线

方法	描述
`fillText()`	在画布上绘制“被填充的”文本
`strokeText()`	在画布上绘制文本（无填充）
`measureText()`	返回包含指定文本宽度的对象

看了上面的表格，相信童鞋们以及有了大概的认识。这一节课，我们先说文本的显示与渲染，用到了 `font` ，`fillText()` 与 `strokeText()` 。剩下三个属性与方法，我们留在后面说一说。

基本文本显示

在Canvas上使用文本，必须得先知道：Canvas上的文本不能使用CSS样式，虽然 `font` 属性与CSS的属性相似，但是却不能够交换使用。

显示文本三步走战略：

1. 使用 `font` 设置字体。
2. 使用 `fillStyle` 设置字体颜色。
3. 使用 `fillText()` 方法显示字体。

这里的 `font` 属性可以不指定，如果没有指定字体，则默认自动使用 10px 无衬线体。

下面的代码简单显示了一段文本，具体属性我们放到后面来说。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>基本文本显示</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

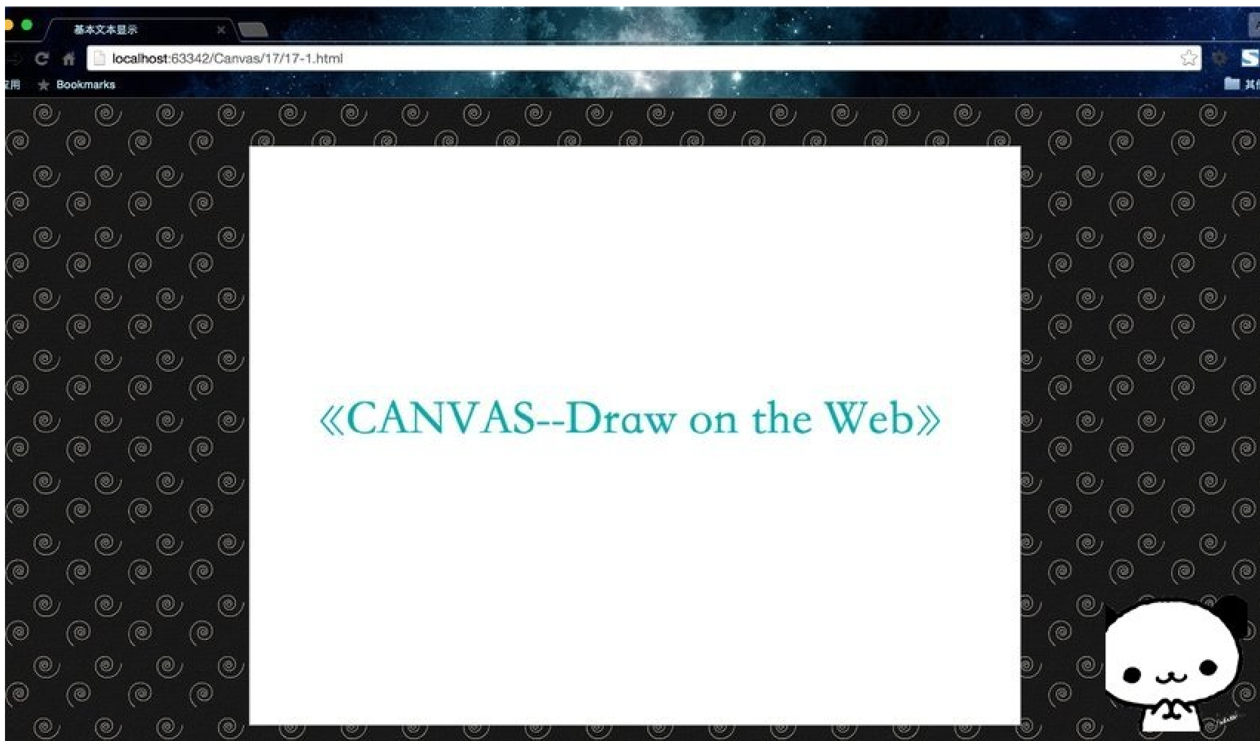
<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        //1. 使用`font`设置字体。
        context.font = "50px serif";
        //2. 使用`fillStyle`设置字体颜色。
        context.fillStyle = "#00AAAA";
        //3. 使用`fillText()`方法显示字体。
        context.fillText("《CANVAS--Draw on the Web》",50,300);

    };
</script>
</body>
</html>
```

演示 17-1

运行结果：



设置文本字体 `font`

在Canvas中设置字体样式非常的容易，`font` 属性与CSS的设置字体格式是一样的，因此只需通过把与CSS兼容的字符串应用到 `font` 属性即可。可以设置字体的样式、字体的变体、字体的粗细、字号和行高、字体外观等。

基本格式如下。

```
context.font =
"[font-style] [font-variant] [font-weight]
[font-size/line-height] [font-family]"
```

以上五个参数均可缺省，各个参数间用逗号隔开。

> 提示：参数用中括号[]包裹起来表示可以缺省。

下面一一来介绍一下这些参数值的意义。

font-style

font-style 属性定义字体的风格。

值	描述
normal	默认值。浏览器显示一个标准的字体样式。
italic	浏览器会显示一个斜体的字体样式。
oblique	浏览器会显示一个倾斜的字体样式。

后两者通常情况下看上去是没啥区别的。但是获取倾斜效果的方法并不同。italic是使用字体库中的斜体字，通常一个字体库是拥有该字体的斜体形式和粗体形式。oblique是直接将字倾斜，如果一个字体库没有斜体字那么就不能使用italic，想要获取倾斜字体只能使用oblique。

font-variant

font-variant 属性设置小型大写字母的字体显示文本，这意味着所有的小写字母均会被转换为大写，但是所有使用小型大写字体的字母与其余文本相比，其字体尺寸更小。

值	描述
normal	默认值。浏览器显示一个标准的字体样式。
small-caps	浏览器会显示小型大写字母的字体。

看下面的一张图片就知道这属性啥意思啦。

查看结果:

CANVAS--Draw on the Web

CANVAS--DRAW ON THE WEB

就是这样，上面一行是使用的默认值normal，下面一行使用的是small-caps。效果就是，原本大写的英文字母不变，小写的英文字母变成大写，但是大小不变。

font-weight

font-weight 属性设置文本的粗细。

值	描述
normal	默认值。浏览器显示一个标准的字体样式。
bold	定义粗体字符。
bolder	定义更粗的字符。
lighter	定义更细的字符。
100-900之间的值	定义由粗到细的字符。400 等同于 normal，而 700 等同于 bold。

font-size

font-size 属性可设置字体的尺寸。

值	描述
xx-small	最小字体。
x-small	较小字体。
small	小字体。
medium	缺省值。
large	大字体。
x-large	较大字体。
xx-large	最大字体。
任意数值	单位px，代表字号值。

line-height

line-height 属性设置行间的距离（行高）。不允许使用负值

font-family

font-family 规定元素的字体系列。

使用 `@font-face` 自定义字体

HTML5支持常用字体，如果没有可以使用 `@font-face` 扩展字体。但是并不建议使用。

@font-face 能够加载服务器端的字体文件，让客户端显示客户端所没有安装的字体。目前支持加载EOT与TTF文件。

示例：代码过长，略。

演示 17-2



这里的字体是我从国内的一个网站——字体下载大宝库中下载的，蛮不错的，如果大家需要啥字体都可以去看看。我这里下载的字体库只有A-Z这26个大写英文字母，所以遇到小写的自动转大写，遇到汉字或者数字它指定用星星符号代替。使用了CSS3中的 `@font-face` 即可自定义字体，是不是非常的酷。

文本渲染

同绘制矩形一样，这里“绘制”文本也给出了两种方法—— `fillText()` 与 `strokeText()` 。之所以说一样，因为这两个方法也可以通过 `fillStyle` 与 `strokeStyle` 设置对应的属性，之前说过的颜色填充、渐变填充、甚至是图案填充都是可以的。

`fillText()` 与 `strokeText()` 的参数表是一样的，接受4个参数，分别是String，x，y与maxlen，其中String是指要显示的字符串，之后x与y是指显示的坐标，最后一个maxlen是可以缺省的数值型参数，代表显示的最大宽度，单位是像素。如果文本的长度超过了这个maxlen，Canvas就会将显示文本横向压缩。通常为了保证字体的美观，我们不设置maxlen。

即 `context.fillText(String,x,y,[maxlen])` 与 `context.strokeText(String,x,y,[maxlen])` 。

下面我们通过一个案例来看看文本渲染的效果。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>文本渲染</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; margin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.beginPath();
        context.font = "50px Verdana";
```

```
        var gradient = context.createLinearGradient(0,0,800,0);
        gradient.addColorStop("0","magenta");
        gradient.addColorStop("0.5","blue");
        gradient.addColorStop("1.0","red");
        context.fillStyle = gradient;
        context.strokeStyle = "#00AAAA";
        context.strokeText("airingursb.github.io", 50, 100);
        context.fillText("airingursb.github.io", 50, 200);

        //限制宽度
        context.fillText("airingursb.github.io", 50, 300, 200);

        context.beginPath();
        var img = new Image();
        img.src = "./images/bg1.jpg";
        img.onload = function(){
            var pattern = context.createPattern(img, "repeat");
            context.fillStyle = pattern;
            context.fillText("airingursb.github.io", 50, 400);
        }

        context.beginPath();
        context.fillStyle = "#00AAAA";
        context.fillText("Airing的博客，欢迎访问", 50, 500);
    };
</script>
</body>
</html>
```

演示 17-3

运行结果：



这里第一行使用的是一般颜色的 `strokeText()` 方法，第二行使用的是渐变色的 `fillText()` 方法，第三行设置了maxlen，第四行给字体填充的是纹理图案，第五行是广告......欢迎访问个人博客！

好了，还有 `font` 那几个参数的不同值大家可以自己试一下，看看粗体是什么效果、斜体什么样子，自己试一试就会有不同的感觉。这节的内容有点多，大家吸收一下，书法之魂已在心中。让我们继续前进！😋

Ch18 文本对齐与度量

文本对齐

水平对齐 `textAlign`

`context.textAlign="center|end|left|right|start";`

其中各值及意义如下表。

值	描述
start	默认。文本在指定的位置开始。
end	文本在指定的位置结束。
center	文本的中心被放置在指定的位置。
left	文本左对齐。
right	文本右对齐。

我们通过一个例子来直观的感受一下。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>textAlign</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>
```

```
<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

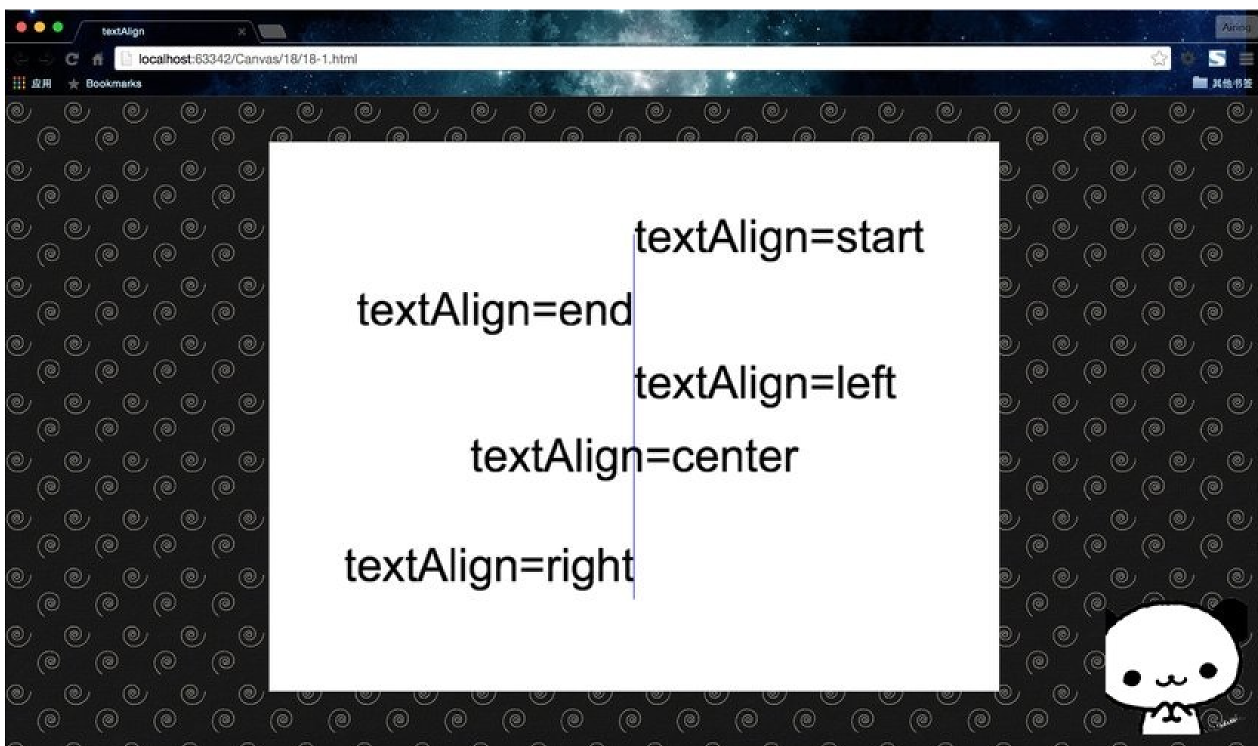
        // 在位置 400 创建蓝线
        context.strokeStyle="blue";
        context.moveTo(400,100);
        context.lineTo(400,500);
        context.stroke();


        context.fillStyle = "#000";
        context.font="50px Arial";

        // 显示不同的 textAlign 值
        context.textAlign="start";
        context.fillText("textAlign=start", 400, 120);
        context.textAlign="end";
        context.fillText("textAlign=end", 400, 200);
        context.textAlign="left";
        context.fillText("textAlign=left", 400, 280);
        context.textAlign="center";
        context.fillText("textAlign=center", 400, 360);
        context.textAlign="right";
        context.fillText("textAlign=right", 400, 480);
    };
</script>
</body>
</html>
```

演示 18-1

运行结果：



垂直对齐 `textBaseline`

```
context.textBaseline="alphabetic|top|hanging|middle|ideographic|bottom";
```

其中各值及意义如下表。

值	描述
alphabetic	默认。文本基线是普通的字母基线。
top	文本基线是 em 方框的顶端。
hanging	文本基线是悬挂基线。
middle	文本基线是 em 方框的正中。
ideographic	文本基线是表意基线。
bottom	文本基线是 em 方框的底端。

首先咱们通过一个图来看一下各个基线代表的位置。



我们通过一个例子来直观的感受一下。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>textBaseline</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
```

```
        context.fillRect(0,0,800,600);

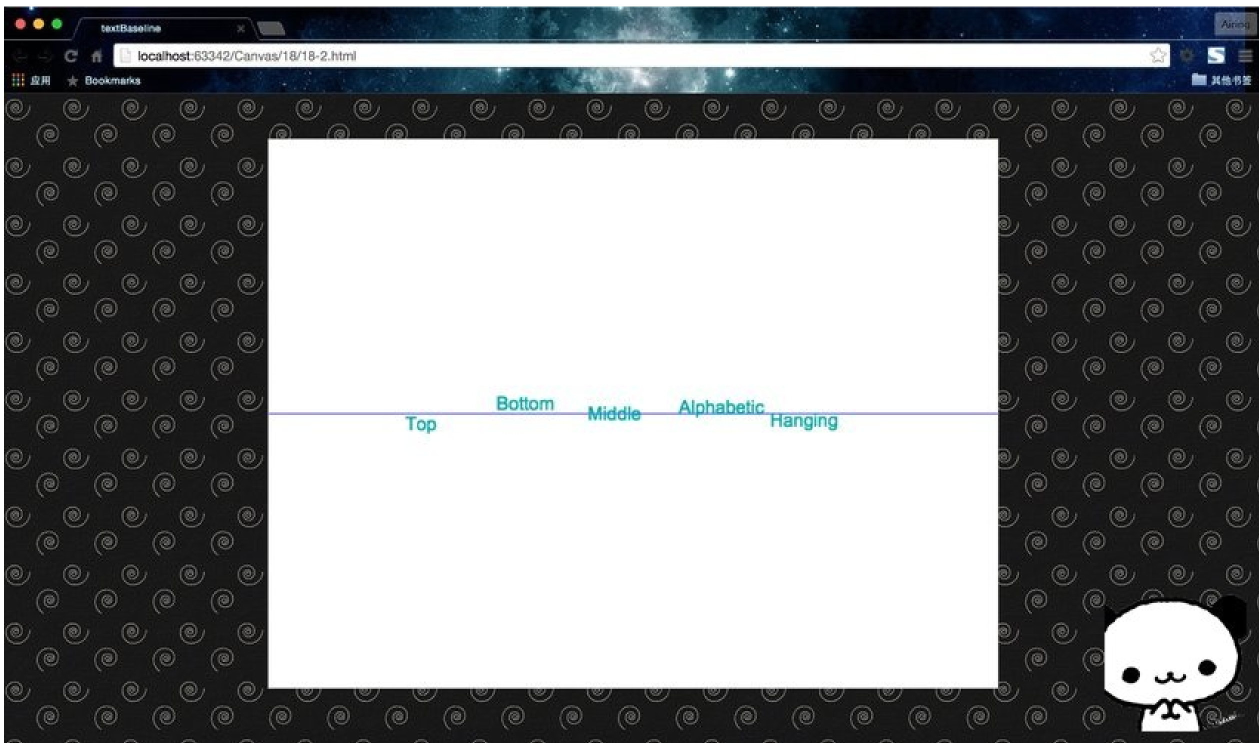
        //在位置 y=300 绘制蓝色线条
        context.strokeStyle="blue";
        context.moveTo(0,300);
        context.lineTo(800,300);
        context.stroke();

        context.fillStyle = "#00AAAA";
        context.font="20px Arial";

        //在 y=300 以不同的 textBaseline 值放置每个单词
        context.textBaseline="top";
        context.fillText("Top",150,300);
        context.textBaseline="bottom";
        context.fillText("Bottom",250,300);
        context.textBaseline="middle";
        context.fillText("Middle",350,300);
        context.textBaseline="alphabetic";
        context.fillText("Alphabetic",450,300);
        context.textBaseline="hanging";
        context.fillText("Hanging",550,300);
    };
</script>
</body>
</html>
```

演示 18-2

运行结果：



文本度量

文本度量使用 `measureText()` 方法实现，这个api在换行显示判断中会有奇效。例如之前提到的微信界面生成器，在对话的字符长度超出一定值的时候，需要换行显示。那么，这个功能需要怎么实现呢？就是通过 `context.measureText(text).width;` 来实现判断。其中，text是要测量的文本。

这里提供一个代码演示一下该方法的作用，大家可以课下自行实现文本换行功能，这个功能是比较实用的，因为Canvas 文本API只支持单行显示。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>measureText</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
```

```
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

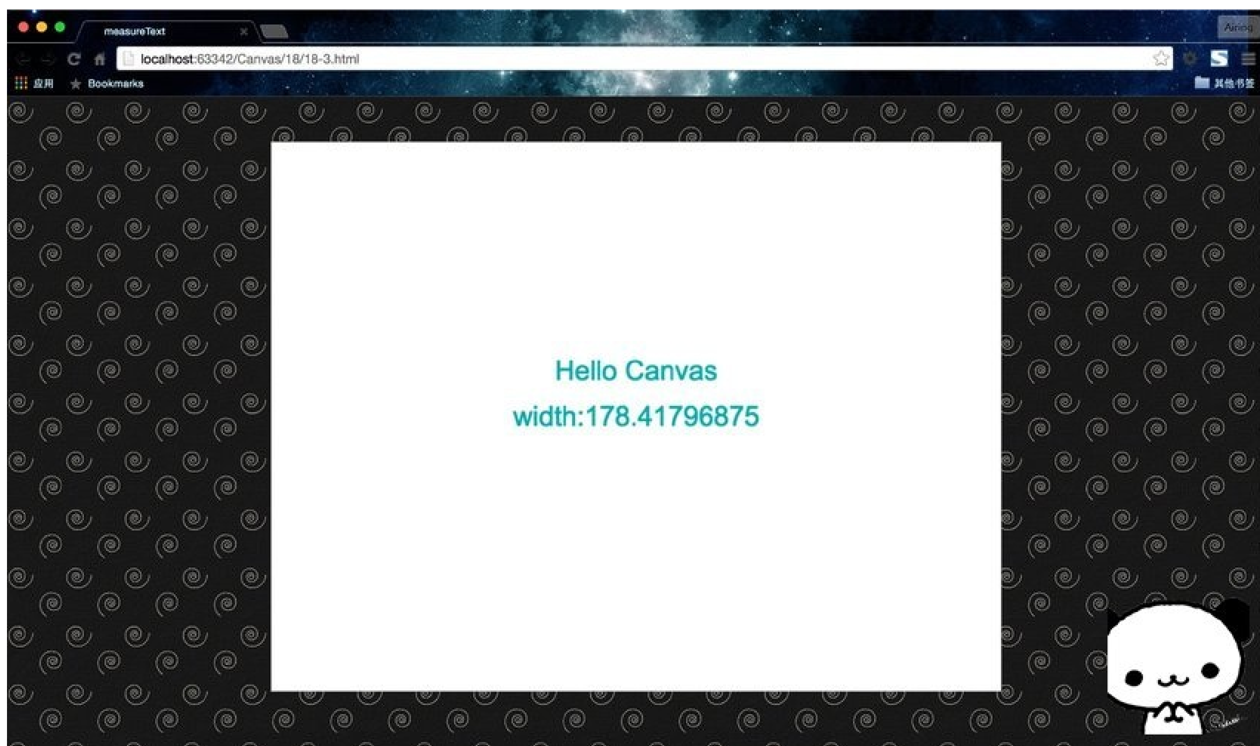
        //居中显示
        context.textAlign = "center";
        context.textBaseline = "middle";

        context.fillStyle = "#00AAAA";
        context.font="30px Arial";
        var txt="Hello Canvas";
        context.fillText("width:" + context.measureText(txt).wid
th,400,300);
        context.fillText(txt,400,250);

    };
</script>
</body>
</html>
```

演示 18-3

运行结果：



至此，Canvas 文本API的内容已经说完了，是不是非常的简单。现在咱们已经有了书法家和艺术家的底蕴了，接下来，咱们学一些Canvas 高级内容API~是不是特别的激动~我们继续前进！😋

Ch19 全局阴影与图像合成

阴影效果

创建阴影效果需要操作以下4个属性：

- `context.shadowColor` ：阴影颜色。
- `context.shadowOffsetX` ：阴影x轴位移。正值向右，负值向左。
- `context.shadowOffsetY` ：阴影y轴位移。正值向下，负值向上。
- `context.shadowBlur` ：阴影模糊滤镜。数据越大，扩散程度越大。

这四个属性只要设置了第一个和剩下三个中的任意一个就有阴影效果。不过通常情况下，四个属性都要设置。

例如，创建一个向右下方位移各5px的红色阴影，模糊2px，可以这样写。

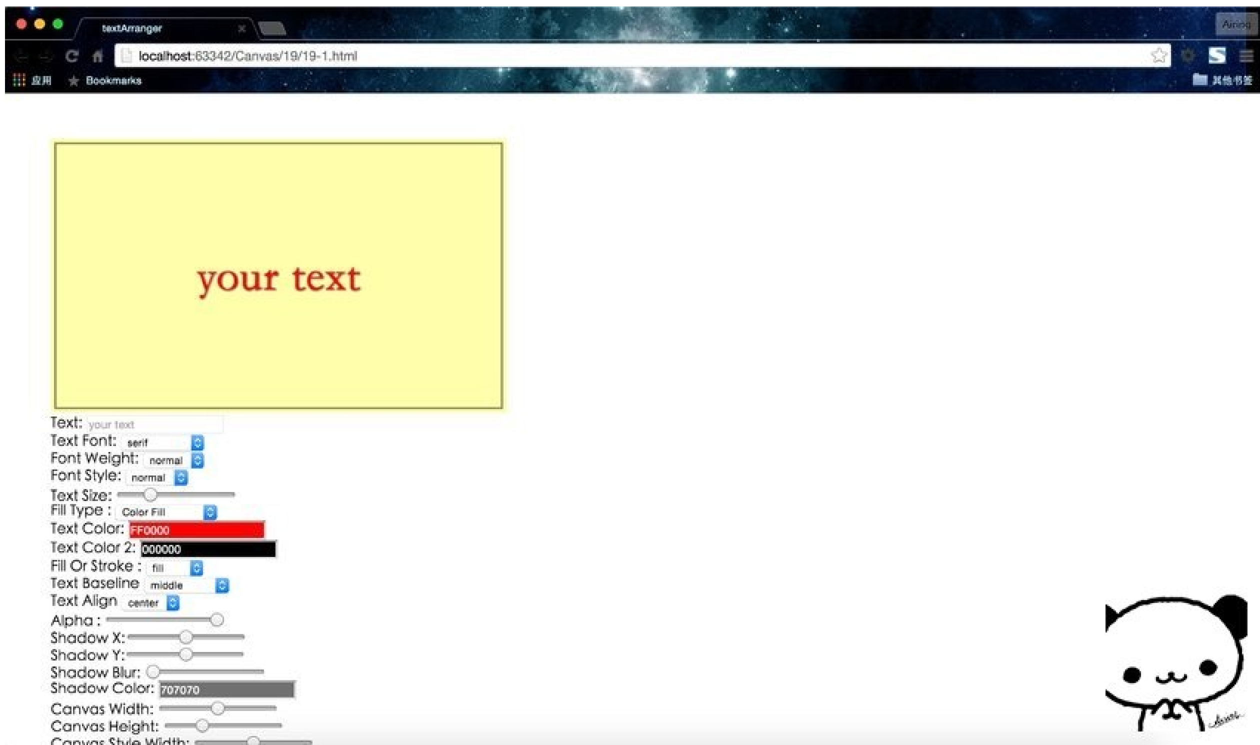
```
context.shadowColor = "red";
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur= 2;
```

需要注意的是，这里的阴影同其他属性设置一样，都是基于状态的设置。因此，如果只想为某一个对象应用阴影而不是全局阴影，需要在下次绘制前重置阴影的这四个属性。

下面的例子摘抄自《HTML5 Canvas开发详解》第二版，案例名为 textArranger，是一个交互的网页。结合了上两节所说的 文本API 和本节的阴影属性。大家可以自己打开链接尝试一下，看看每个属性的效果是什么。本例代码过长，在此不贴。

演示 19-1

运行结果：



全局透明 `globalAlpha`

这个也是很简单的一个属性，默认值为1.0，代表完全不透明，取值范围是0.0（完全透明）~1.0。这个属性与阴影设置是一样的，如果不想针对全局设置不透明度，就得在下次绘制前重置 `globalAlpha` 。

> 总结一下：基于状态的属性有哪些？
>
> —— `globalAlpha`
>
> —— `globalCompositeOpeartion`
>
> —— `strokeStyle`
>
> —— `textAlign` , `textBaseline`
>
> —— `lineCap` , `lineJoin` , `lineWidth` , `miterLimit`
>
> —— `fillStyle`
>
> —— `font`
>
> —— `shadowBlur` , `shadowColor` , `shadowOffsetX` , `shadowOffsetY`

我们通过一个代码，来体验一下 `globalAlpha` 的神奇之处~

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>全局透明</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.globalAlpha = 0.5;

        for(var i=0; i<=50; i++){
            var R = Math.floor(Math.random() * 255);
            var G = Math.floor(Math.random() * 255);
            var B = Math.floor(Math.random() * 255);

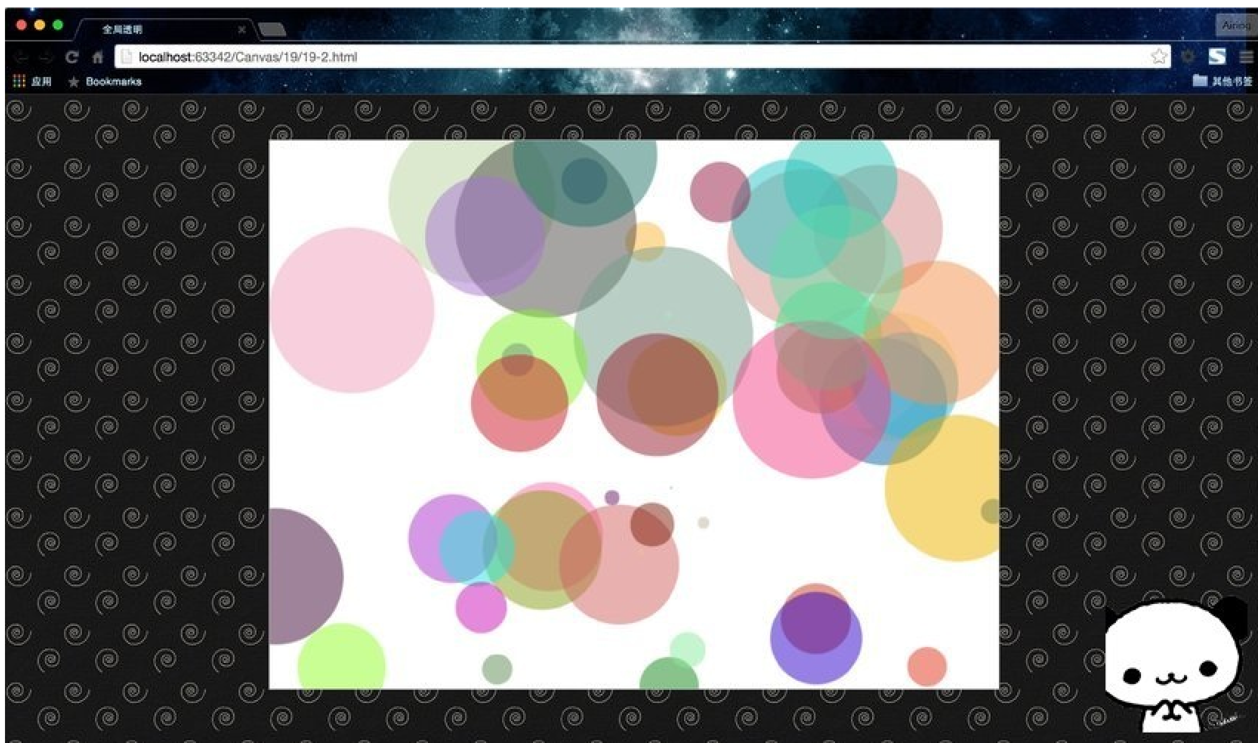
            context.fillStyle = "rgb(" + R + "," + G + "," + B +
")";

            context.beginPath();
            context.arc(Math.random() * canvas.width, Math.rando
m() * canvas.height, Math.random() * 100, 0, Math.PI * 2);
```

```
                context.fill();
            }
        };
</script>
</body>
</html>
```

演示 19-2

运行结果：



是不是非常的酷？终于有点艺术家的范儿了吧。

图像合成 `globalCompositeOperation`

两个图像重合的时候，就涉及到了对这两个图像的合成处理。`globalCompositeOperation` 属性设置或返回如何将一个源（新的）图像绘制到目标（已有）的图像上。

源图像 = 您打算放置到画布上的绘图。

目标图像 = 您已经放置在画布上的绘图。

值	描述
source-over	默认。在目标图像上显示源图像。
source-atop	在目标图像顶部显示源图像。源图像位于目标图像之外的部分是不可见的。
source-in	在目标图像中显示源图像。只有目标图像内的源图像部分会显示，目标图像是透明的。
source-out	在目标图像之外显示源图像。只会显示目标图像之外源图像部分，目标图像是透明的。
destination-over	在源图像上方显示目标图像。
destination-atop	在源图像顶部显示目标图像。源图像之外的目标图像部分不会被显示。
destination-in	在源图像中显示目标图像。只有源图像内的目标图像部分会被显示，源图像是透明的。
destination-out	在源图像外显示目标图像。只有源图像外的目标图像部分会被显示，源图像是透明的。
lighter	显示源图像 + 目标图像。
copy	显示源图像。忽略目标图像。
xor	使用异或操作对源图像与目标图像进行组合。

下面我用一段代码来说明一下这些值的意义。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>图像合成</title>
    <style>
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
```

```
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");

        context.globalCompositeOperation = "source-out";
        context.globalAlpha = 0.5;

        for(var i=0; i<=50; i++){
            var R = Math.floor(Math.random() * 255);
            var G = Math.floor(Math.random() * 255);
            var B = Math.floor(Math.random() * 255);

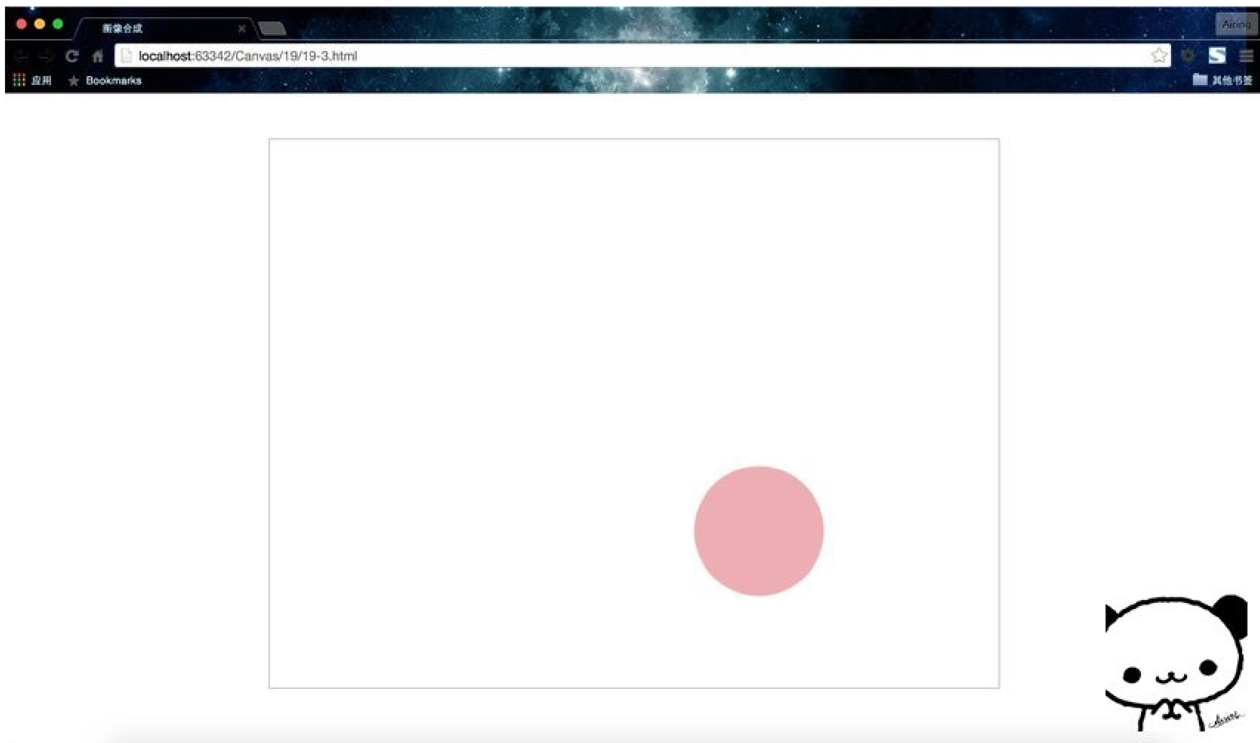
            context.fillStyle = "rgb(" + R + "," + G + "," + B +
")";

            context.beginPath();
            context.arc(Math.random() * canvas.width, Math.rando
m() * canvas.height, Math.random() * 100, 0, Math.PI * 2);
            context.fill();
        }

    };
</script>
</body>
</html>
```

演示 19-3

运行结果：



我这个代码相对比较简单，这里推荐一下laijieyao的专栏——【HTML5】Canvas之globalCompositeOperation属性详解，这篇博客里面介绍了该属性的11值的不同效果，大家可以看一下有一个直观的感受。

这里的代码和上面的一样，但是用到了图像合成，所以把背景和空白矩形去掉了，不然的话所有的图像都是合成图像，因为他们都是建立在那个空白矩形之上的。虽然只加了 `context.globalCompositeOperation = "source-out";` 一句而已，其他的圆都不见了，有时候只会显示1~2个圆。因为source-out就是图像合成处理时，保留没有被叠加的图像。因此可以推出，在随机生成的50个圆中，有49个都重叠了。

还有其他的属性，童鞋们私下也可以自己尝试一下，都能带来不一样的神奇效果。

这一小节介绍了阴影、透明与图像合成，大家消化一下，建议其他属性值都自己尝试一下加深理解。我们继续向着终点前进！~😊

Ch20 裁剪和绘制图像

裁剪区域 `clip()`

使用Canvas绘制图像的时候，我们经常会想要只保留图像的一部分，这是我们可以使用canvas API再带的图像裁剪功能来实现这一想法。

Canvas API的图像裁剪功能是指，在画布内使用路径，只绘制该路径内所包含区域的图像，不绘制路径外的图像。这有点像Flash中的图层遮罩。

使用图形上下文的不带参数的 `clip()` 方法来实现Canvas的图像裁剪功能。该方法使用路径来对Canvas画布设置一个裁剪区域。因此，必须先创建好路径。创建完整后，调用 `clip()` 方法来设置裁剪区域。

需要注意的是裁剪是对画布进行的，裁切后的画布不能恢复到原来的大小，也就是说画布是越切越小的，要想保证最后仍然能在canvas最初定义的大小下绘图需要注意 `save()` 和 `restore()` 。画布是先裁切完了再进行绘图。并不一定非要是图片，路径也可以放进去~

先来看看一个简单的Demo。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>裁剪区域</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
```

```
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        //在屏幕上绘制一个大方块
        context.fillStyle = "black";
        context.fillRect(10,10,200,200);
        context.save();
        context.beginPath();

        //裁剪画布从(0，0)点至(50，50)的正方形
        context.rect(0,0,50,50);
        context.clip();

        //红色圆
        context.beginPath();
        context.strokeStyle = "red";
        context.lineWidth = 5;
        context.arc(100,100,100,0,Math.PI * 2,false);
        //整圆
        context.stroke();
        context.closePath();

        context.restore();

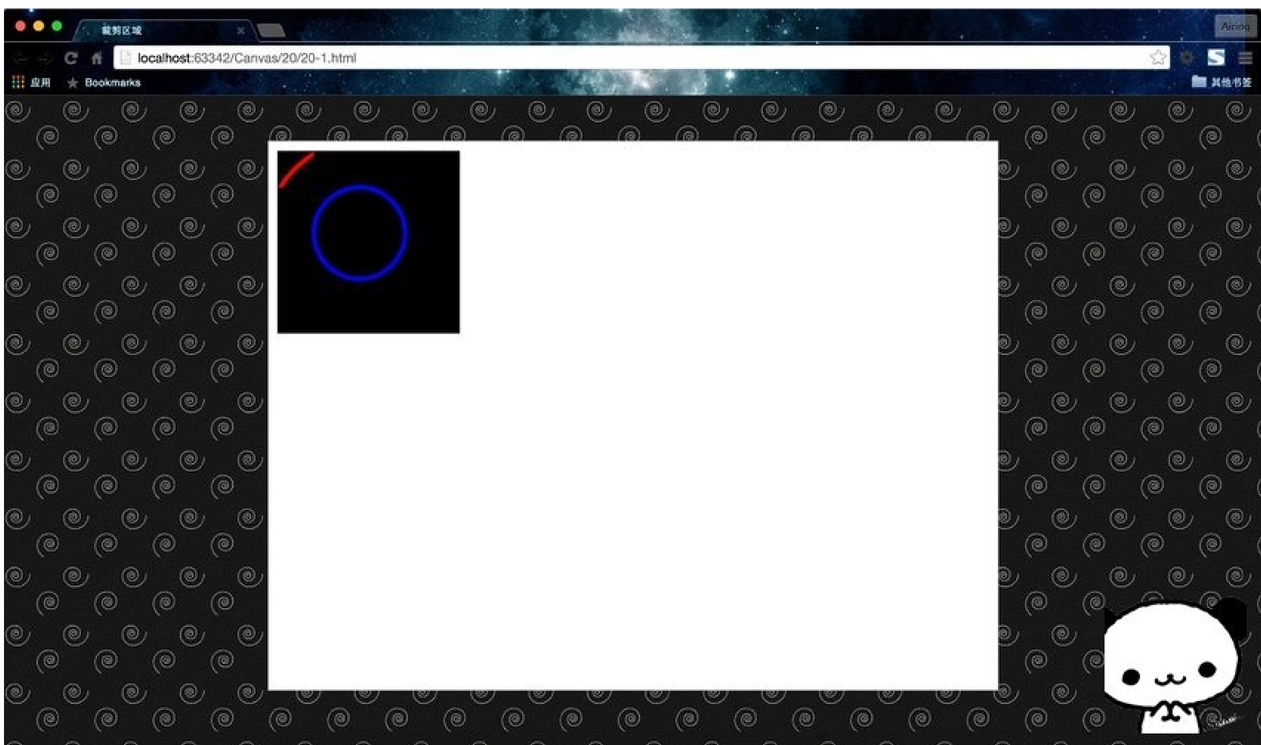
        //再次裁切整个画布
        context.beginPath();
        context.rect(0,0,500,500);
        context.clip();

        //绘制一个没有裁切的蓝线
        context.beginPath();
        context.strokeStyle = "blue";
```

```
        context.lineWidth = 5;
        context.arc(100,100,50,0,Math.PI * 2,false);
        //整圆
        context.stroke();
        context.closePath();
    };
</script>
</body>
</html>
```

演示 20-1

运行结果：



自己分析吧，能够理解这段程序，就完全掌握了 `clip()` 方法的使用了。

绘制图像 `drawImage()`

`drawImage()` 是一个很关键的方法，它可以引入图像、画布、视频，并对其进行缩放或裁剪。

一共有三种表现形式：

1. 三参数：`context.drawImage(img,x,y)`

2. 五参数：`context.drawImage(img,x,y,width,height)`
3. 九参数：`context.drawImage(img,sx,sy,swidth,sheight,x,y,width,height)`

三参数的是标准形式，可用于加载图像、画布或视频；五参数的除了可以加载图像还可以对图像进行指定宽高的缩放；九参数的除了缩放，还可以裁剪。各参数意义见下表。

参数	描述
img	规定要使用的图像、画布或视频。
sx	可选。开始剪切的 x 坐标位置。
sy	可选。开始剪切的 y 坐标位置。
swidth	可选。被剪切图像的宽度。
sheight	可选。被剪切图像的高度。
x	在画布上放置图像的 x 坐标位置。
y	在画布上放置图像的 y 坐标位置。
width	可选。要使用的图像的宽度。（伸展或缩小图像）
height	可选。要使用的图像的高度。（伸展或缩小图像）

下面，我们加载一个图片试试。

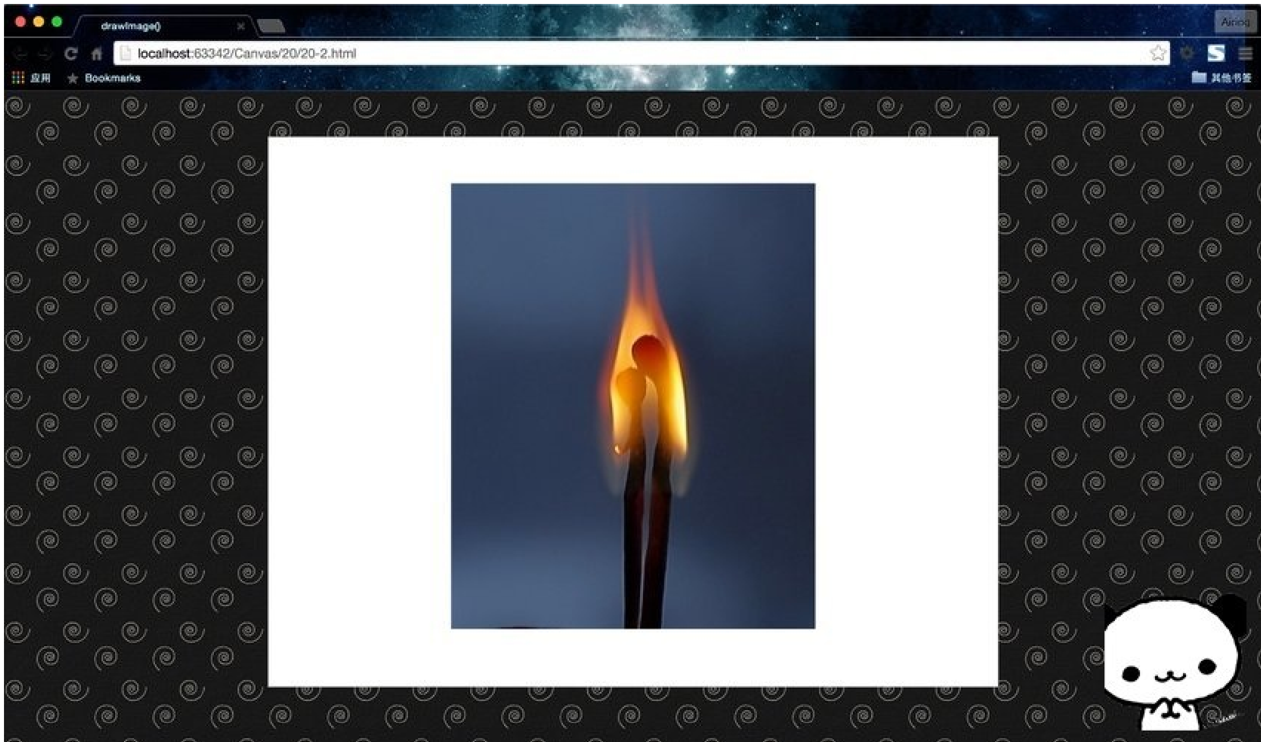
```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>drawImage()</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        var img = new Image();
        img.src = "./images/20-1.jpg";
        img.onload = function(){
            context.drawImage(img,200,50);
        }
    };
</script>
</body>
</html>
```

演示 20-2

运行结果：



创建相框

这里，我们结合 `clip()` 和 `drawImage()` 以及之前学的三次贝塞尔曲线 `bezierCurveTo()` ，来为上面一个案例，加上一个心形的相框~

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>绘制心形相框</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; margin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
```

```
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.beginPath();
        context.moveTo(400,260);
        context.bezierCurveTo(450,220,450,300,400,315);
        context.bezierCurveTo(350,300,350,220,400,260);
        context.clip();
        context.closePath();

        var img = new Image();
        img.src = "./images/20-1.jpg";
        img.onload = function(){
            context.drawImage(img,348,240,100,100);
        }
    };
</script>
</body>
</html>
```

演示 20-3

运行结果截图：



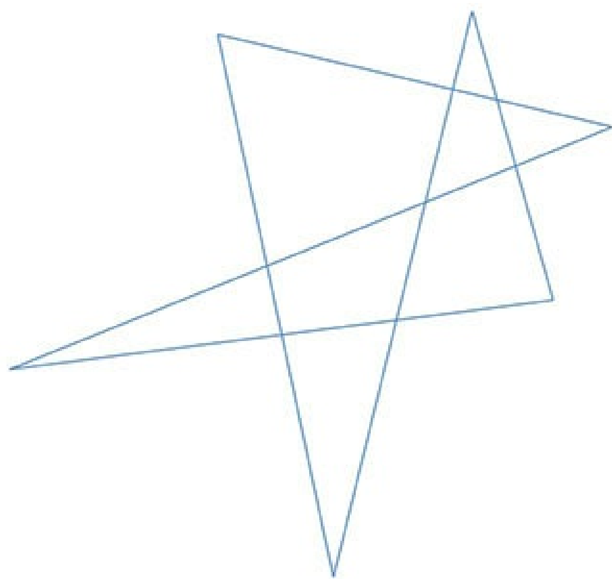
是不是美美的？好啦，至此最关键的遮罩和图像裁剪已经说完了，其实在java.awt中，`drawImage()`也是一个至关重要的方法。有人说制作Java游戏界面，只要会用`drawImage()`就可以一招打遍天下~在Canvas里也是一样的。美工提供的素材基本都是图片，这个时候`drawImage()`对图片的处理就很重要了。既是基本功，也是对图片最重要的处理方法。

这一节就跟大家吹这么多了~让我们继续前进！😋

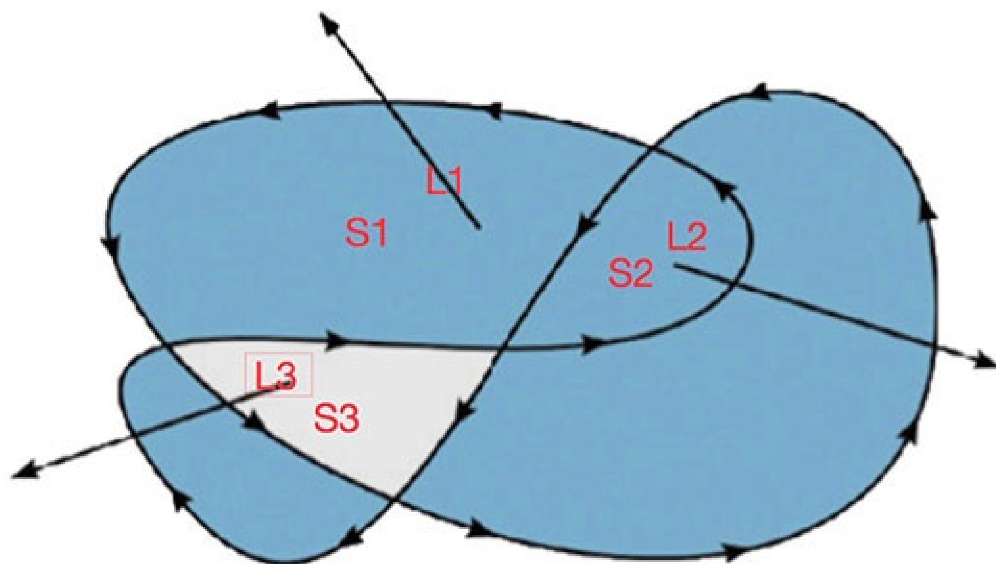
Ch21 非零环绕原则

路径方向与非零环绕原则

平时我们画的图形都是规规矩矩的，那么如果我们用线条画了个抽象派作品，就像下面这图一样，童鞋们知道怎么用 `fill()` 染色呢？



这里就要用到数学上的一个方法——非零环绕原则，来判断哪块区域是里面，哪块区域是外面。接下来，我们具体来看下什么是非零环绕原则。



首先，我们得给图形确定一条路径，只要“一笔画”并且“不走重复路线”就可以了。如图，标出的是其中的一种路径方向。我们先假定路径的正方向为1（其实为-1啥的也都可以，正负方向互为相反数，不是0就行），那么反方向就是其相反数-1。

然后，我们在子路径切割的几块区域内的任意一点各取一条方向任意的射线，这里我只取了三个区域的射线为例，来判断这三块区域是“里面”还是“外面”。

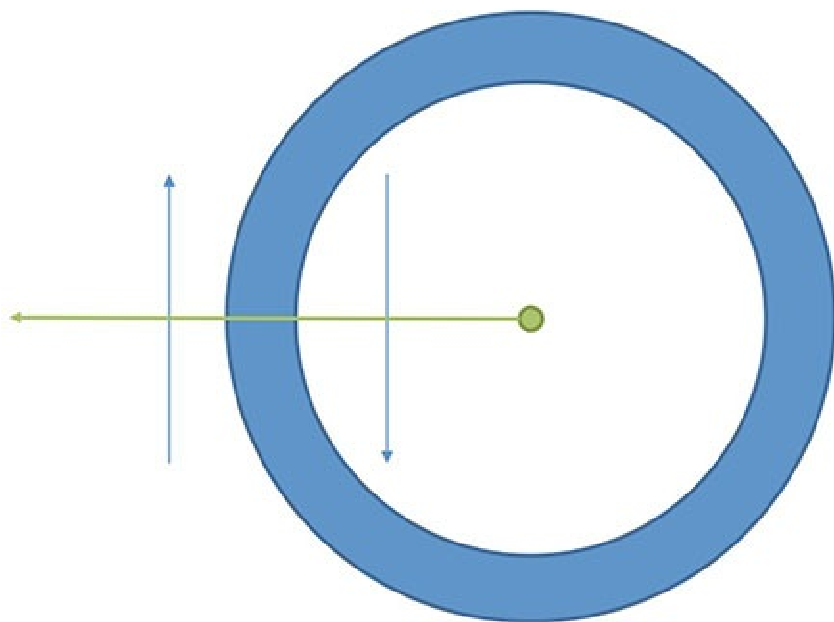
接下来，我们就来判断了。S1中引出的射线L1，与S1的子路径的正方向相交，那么我们就给计数器+1，结果为+1，在外面。

S2中引出的射线L2，与两条子路径的正方向相交，计数器+2，结果为+2，在外面。

S3中引出的射线L3，与两条子路径相交，但是其中有一条的反方向，计数器+1-1，结果为0，在里面。没错，只要结果不为0，该射线所在的区域就在外面。

绘制圆环

记得我们之前学过的 `arc` 方法吗？它的最后一个参数就是判断是路径方向的，如果是路径相反的两个同心圆在一起，图上色会有什么神奇的效果呢？



下面我们通过代码来实现它。

```
<!DOCTYPE html>
<html lang="zh">
<head>
```

```
    <meta charset="UTF-8">
    <title>圆环</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

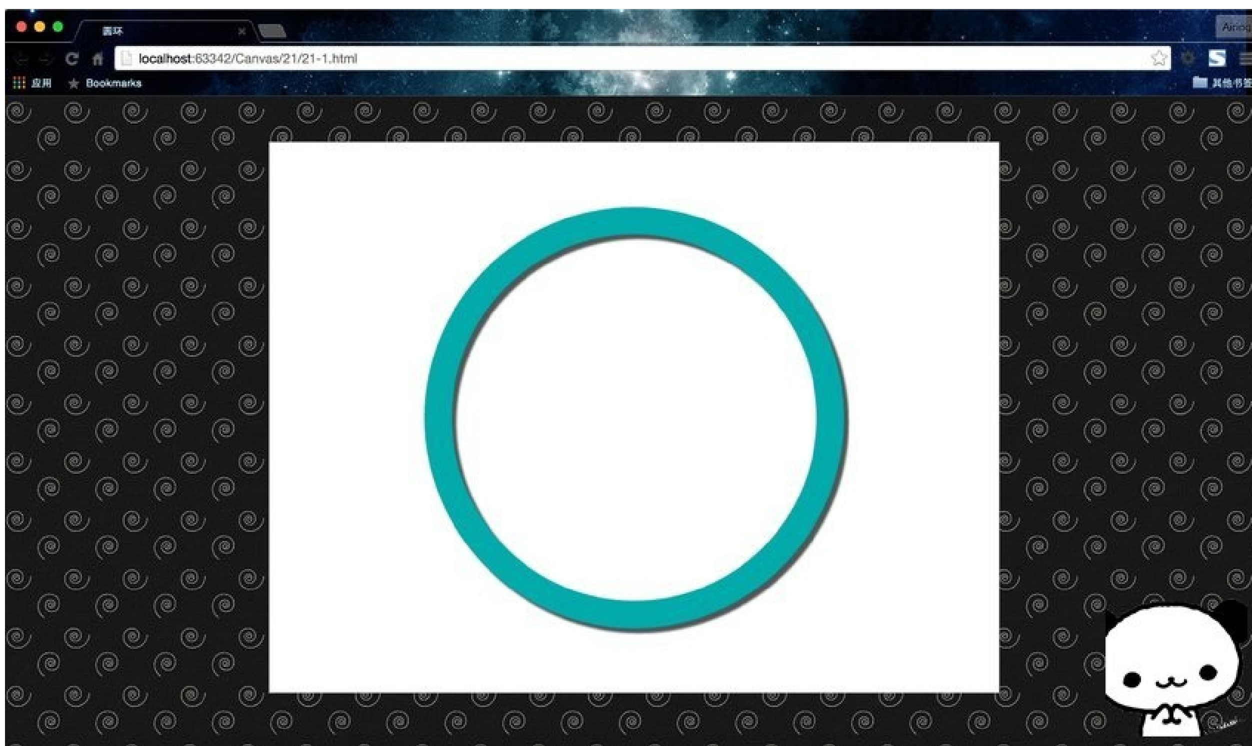
<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.shadowColor = "#545454";
        context.shadowOffsetX = 5;
        context.shadowOffsetY = 5;
        context.shadowBlur = 2;

        context.arc(400, 300, 200, 0, Math.PI * 2 ,false);
        context.arc(400, 300, 230, 0, Math.PI * 2 ,true);
        context.fillStyle = "#00AAAA";
        context.fill();
    };
</script>
</body>
</html>
```

演示 21-1

运行结果：



结合我们上一节学到了阴影效果，这个圆环看上去是不是特别的有立体感？

镂空剪纸效果

接下来，我们利用非零环绕原则和阴影来绘制一个镂空的剪纸效果。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>镂空剪纸效果</title>
    <style>
        body { background: url("./images/bg3.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
```

```
    </canvas>
</div>

<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        context.beginPath();
        context.rect(200,100,400,400);
        drawPathRect(context, 250, 150, 300, 150);
        drawPathTriangle(context, 345, 350, 420, 450, 270, 450);
        context.arc(500, 400, 50, 0, Math.PI * 2, true);
        context.closePath();

        context.fillStyle = "#058";
        context.shadowColor = "gray";
        context.shadowOffsetX = 10;
        context.shadowOffsetY = 10;
        context.shadowBlur = 10;
        context.fill();

    };

    //逆时针绘制矩形
    function drawPathRect(cxt, x, y, w, h){
        /**
         * 这里不能使用beginPath和closePath，
         * 不然就不属于子路径而是另一个全新的路径，
         * 无法使用非零环绕原则
         */
        cxt.moveTo(x, y);
        cxt.lineTo(x, y + h);
        cxt.lineTo(x + w, y + h);
        cxt.lineTo(x + w, y);
        cxt.lineTo(x, y);
```

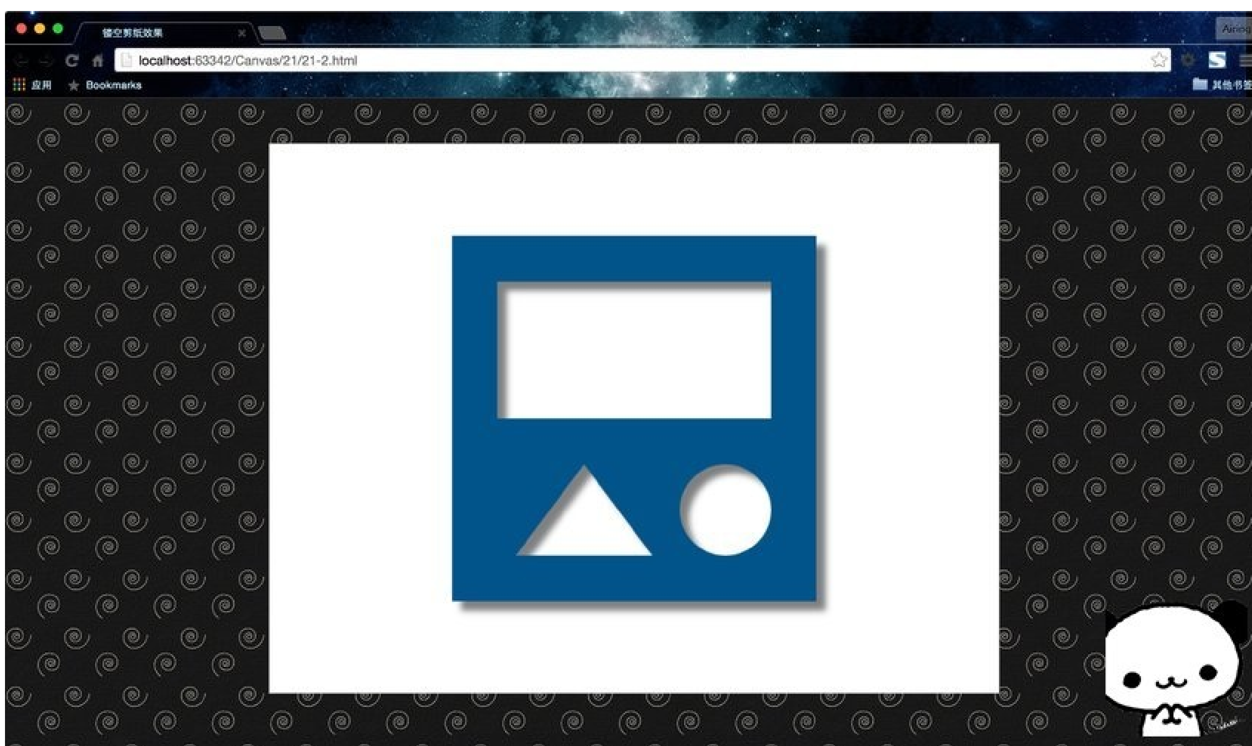
```
    }

    //逆时针绘制三角形
    function drawPathTriangle(cxt, x1, y1, x2, y2, x3, y3){
        cxt.moveTo(x1,y1);
        cxt.lineTo(x3,y3);
        cxt.lineTo(x2,y2);
        cxt.lineTo(x1,y1);
    }

</script>
</body>
</html>
```

演示 21-3

运行结果：



这里手动绘制矩形的原因是我们想要得到逆时针路径的矩形，而且API提供的 `rect()` 方法绘制是顺时针矩形。另外，需要注意的是，这个剪纸是一个图形，一个路径。不能在绘制镂空三角形和绘制镂空矩形的方法里使用 `beginPath()` 和 `closePath()` ，不然它们就会是新的路径、新的图形，而不是剪纸的子路径、子图形，就无法使用非零环绕原则。

好了，这一节的内容就到这里，内容相对来说还是比较简单实用的。下一节就是Canvas API的最后一节了，大家已经掌握了这么多的绘制方法，是不是跃跃欲试了呢？那么，就扬起手中的笔，绘出自己的艺术家之魂吧~😋

Ch22 最后的API

橡皮擦 `clearRect()`

之前一直教大家怎么绘图，各种画笔各种样式，却没有教过童鞋们使用橡皮擦。Canvas 提供了 `clearRect()` 方法，就是清空指定矩形上的画布上的像素。它接受四个参数，和其他绘制矩形的方法一样—— `context.clearRect(x,y,w,h)` 。

下面，我们把之前新画布（实例 9-1）上的空白矩形给擦了吧！让整个页面显示出完整的背景图片。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>clearRect()</title>
    <style>
        body { background: url("./images/bg2.jpg") repeat; }
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>

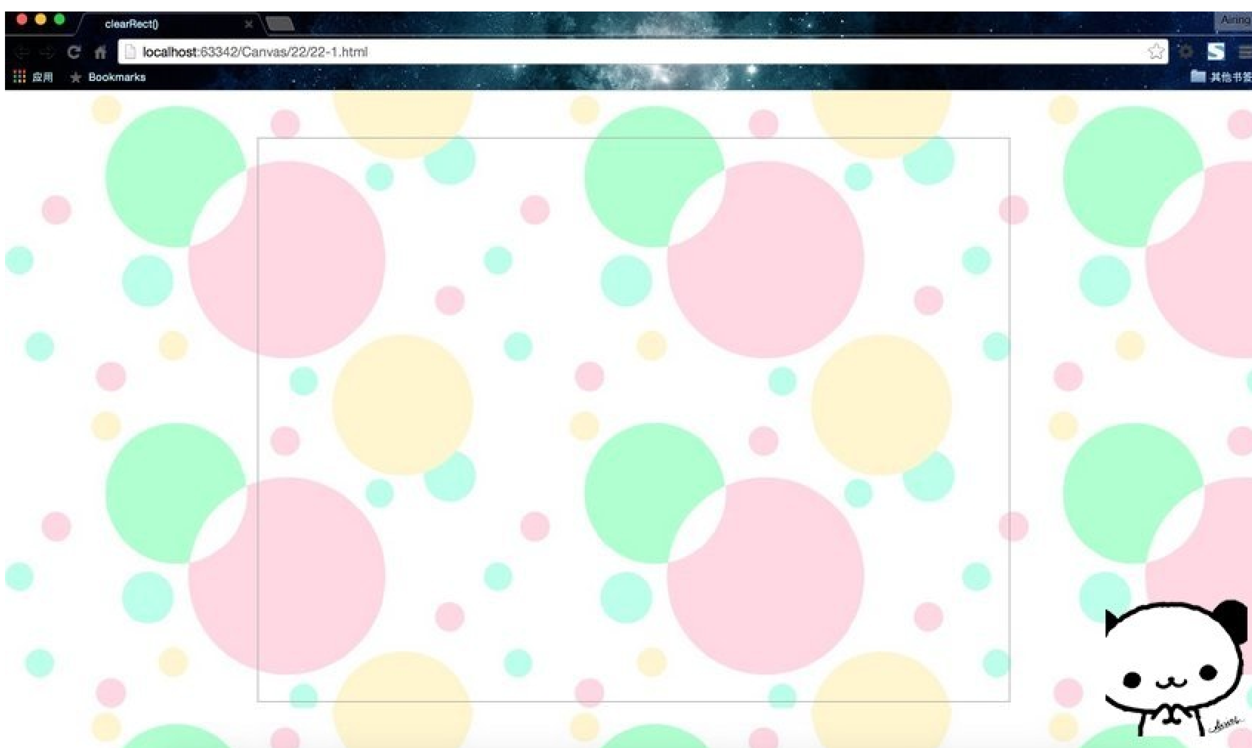
<script>
    window.onload = function(){
        var canvas = document.getElementById("canvas");
        canvas.width = 800;
        canvas.height = 600;
        var context = canvas.getContext("2d");
        context.fillStyle = "#FFF";
        context.fillRect(0,0,800,600);

        //清空画布
        context.clearRect(0,0,canvas.width,canvas.height);

    };
</script>
</body>
</html>
```

演示 22-1

运行结果：



橡皮擦就是这么简单~

点泡泡小游戏

这里通过一个小游戏介绍一个交互性很强的API—— `isPointInPath()` 。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>点泡泡</title>
    <style>
        #canvas { border: 1px solid #aaaaaa; display: block; mar
gin: 50px auto; }
    </style>
</head>
<body>
<div id="canvas-warp">
    <canvas id="canvas">
        你的浏览器居然不支持Canvas？！赶快换一个吧！！
    </canvas>
</div>
```

```
<script>
    var balls = [];
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");

    window.onload = function(){
        canvas.width = 800;
        canvas.height = 600;

        for(var i=0; i<50; i++){
            var aBall = {
                x: Math.random() * canvas.width,
                y: Math.random() * canvas.height,
                r: Math.random() * 50 + 20
            };
            balls[i] = aBall;
        }

        draw();
        canvas.addEventListener("mousemove",detect);
    };

    function draw(){
        for(var i=0; i<balls.length; i++){
            context.beginPath();
            context.arc(balls[i].x, balls[i].y, balls[i].r, 0, M
ath.PI *2);
            context.globalAlpha = 0.5;

            var R = Math.floor(Math.random() * 255);
            var G = Math.floor(Math.random() * 255);
            var B = Math.floor(Math.random() * 255);

            context.fillStyle = "rgb(" + R + "," + G + "," + B +
")";
            context.fill();

        }
    }
```

```
    function detect(){
        var x = event.clientX - canvas.getBoundingClientRect().l
eft;
        var y = event.clientY - canvas.getBoundingClientRect().t
op;

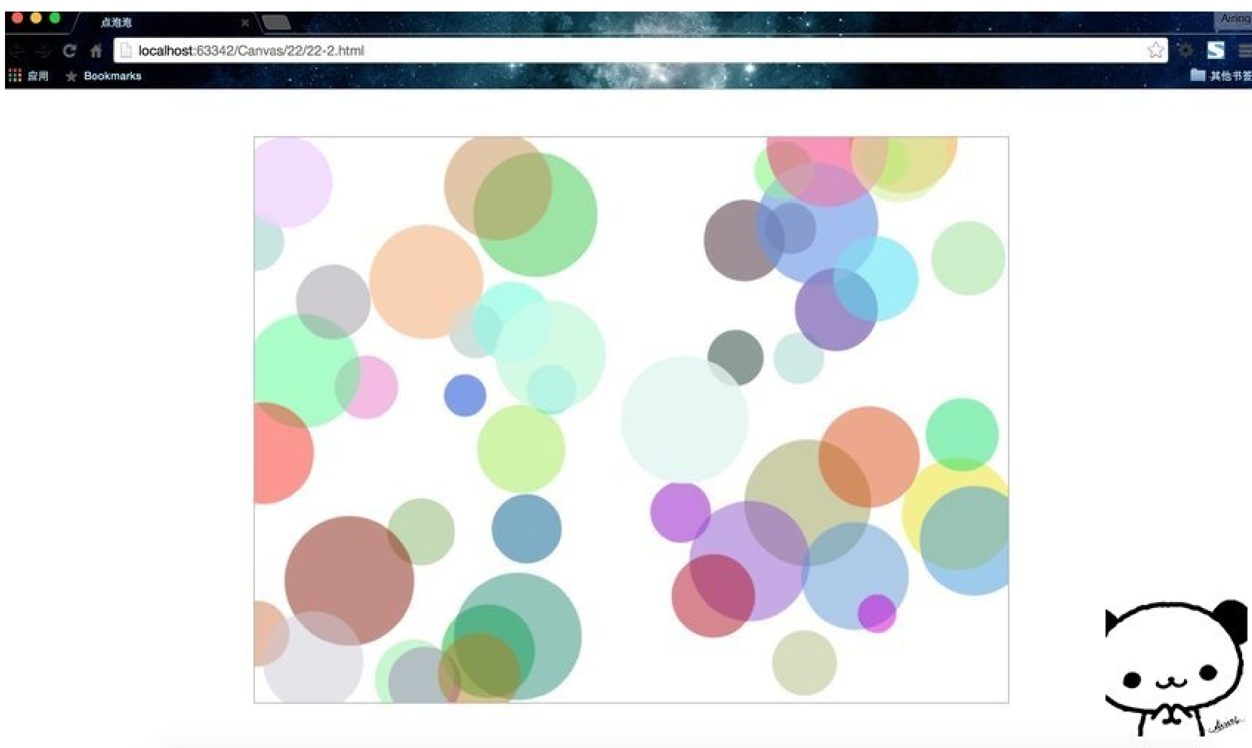
        for(var i=0; i<balls.length; i++){
            context.beginPath();
            context.arc(balls[i].x, balls[i].y, balls[i].r, 0, M
ath.PI * 2);

            if(context.isPointInPath(x,y)){
                context.fillStyle = "rgba(255,255,255,0.1)";
                context.fill();
            }
        }
    }

</script>
</body>
</html>
```

演示 22-2

运行结果：



这个是基于示例 19-2小的交互游戏。鼠标移动到小球上，小球就会渐渐消失。这里用到了鼠标事件 `canvas.addEventListener("mousemove",function);` 以后会详细说。还用到了一个新的API—— `isPointInPath()` 。这个方法接收两个参数，就是一个点的坐标值，用来判断指定的点是否在当前路径中。若是，则返回true。

像素操作API

还有最后六个关于像素操作的API，基本不会用到，这里就不详细说了。列表如下。

属性	描述
`width`	返回 `ImageData` 对象的宽度
`height`	返回 `ImageData` 对象的高度
`data`	返回一个对象，其包含指定的 `ImageData` 对象的图像数据

方法	描述
`createImageData()`	创建新的、空白的 `ImageData` 对象
`getImageData()`	返回 `ImageData` 对象，该对象为画布上指定的矩形复制像素数据
`putImageData()`	把图像数据（从指定的 `ImageData` 对象）放回画布上

如果童鞋们想深入学习，可以直接查HTML5 Canvas参考手册，自行了解学习。

Canvas 图形库

不知不觉写了22个章节，所有我们写的图形其实都可以封装在一个JS文件里，这个文件就是属于我们自己的图形库、图形引擎。当然，第三方也提供了很多优秀的图形库，这里推荐三个给大家。

1. canvasplus
2. Artisan JS
3. Rgraph

大家有兴趣的话可以自行查阅了解一下。

Canvas API没有结束

Canvas的标准一直在更新，大家可以访问 W3C Canvas标准查看最新的API。但是一般最新的API很多浏览器都不会立刻去支持，所以可以等待大多数浏览器稳定支持了之后，我们再去掌握它也不迟。

至此，目前所有的Canvas API我们就已经讲完了。掌握了所有的绘画方法和技巧，成为一个艺术家，并不是我们最终的目标。或许，现在大家已然可以绘制出优美的图形，或抽象、或清新。或许，现在大家已经可以将Canvas API铭记于心，并且能够熟练使用它。

但是要知道，这只是基础。在之后的日子里，我们要基于Canvas 学习动画。众所周知，动画是由一帧帧的画面构成，不会绘画哪来动画？所以，Canvas绘制只是后面学习的基础。

这不是结束，而是一个新的开始。让我们继续前进~😋